MASL, langage de contrôle multi-agents robotiques

Michel Dubois

THESE / UNIVERSITE DE BRETAGNE-SUD
sous le sceau de l'Université Européenne de Bretagne

pour obtenir le titre de :
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE-SUD

Mention : Sciences et Technologies de l'Information et de la Communication
École Doctorale SICMA

présentée par
Michel Dubois

Préparée à l'Équipe d'Accueil 2593
Laboratoire de Recherche en Informatique
et ses Applications de Vannes et Lorient (VALORIA)

N° d'ordre : 133

MASL, langage de contrôle multi-agents robotiques

Thèse soutenue le 8 décembre 2008
devant le jury composé de :

Philippe Lucidarme
Maître de conférences Université d'Angers / *président*

Jacques Ferber
Professeur Université Montpellier 2 / *rapporteur*

Laurent Nana
Professeur Université de Bretagne Occidentale / *rapporteur*

Dominique Duhaut
Directeur de thèse

Yann Le Guyadec
Co-encadrant de thèse

Remerciements

J’adresse mes remerciements à messieurs Jacques Ferber et Laurent Nana qui ont accepté d’être rapporteur de ce mémoire et pour le grand soin et le temps consacrés à cette tâche. Mes remerciements vont aussi à Philippe Lucidarme qui m’a fait l’honneur de présider le jury de soutenance. Je leur sais gré des conseils et remarques qu’ils m’ont donnés à cette occasion.

Je remercie Dominique Duhaut, directeur de ma thèse et Yann Le Guyadec, qui a encadré mes travaux. Ils ont eu la patience de m’accompagner durant les six longues années de cette thèse. Si faire une thèse en travaillant est un exercice certainement difficile, l’encadrer n’est pas de tout repos. Ces travaux ont été fait en parallèle d’un service d’enseignement du second degré à fort caractère technologique, d’activités collectives et obligation de service (mise en place du c2i), et enfin d’activités techniques (administration de la plateforme SAS V9, mise en place de la plateforme BusinessObjects). Pour leur complémentarité dans l’encadrement, leur enthousiasme, leur intérêt, leur rigueur et encore une fois leur patience, mille fois merci à Dominique et à Yann.

Je souhaite remercier Frank Poirier et Pierre-François Marteau, directeurs successifs du laboratoire, ainsi que sa secrétaire, Sylvianne Boisadan, de m’avoir accueilli. Je les remercie pour tous les moyens mis à ma disposition. Mes pensées vont aussi à toute l’équipe de chercheur et doctorants du très convivial VALORIA.

Je remercie les membres du projet ROBEA MAAM qui, par leurs interactions, m’ont permis de préciser mon projet de recherche dans les premières années. Je pense notamment à Frédéric Lemenn, Faïz Ben Amar et Philippe Bidaud du LRP/Paris, à Simon Denier, Jacques Malenfant et à Alexis Drogoul du LIP6/Paris 6 et à Serge Stinckwich du GREYC/Caen. Je remercie plus particulièrement Claude Gueganno et Elian Carillo qui, également doctorants de mon équipe RIMH, étaient en première ligne.

Je remercie mes collègues du département STID de l’IUT de vannes pour leurs encouragements.

Je remercie les membres de l’ADEFOPE pour les aménagements de service consentis pour me permettre de terminer cette thèse.

Je souhaite également remercier Karen Le Dessert pour le suivi administratif souple de ce travail. Je remercie le conseil scientifique de l’Université de Bretagne-Sud pour les deux décharges de service qu’il a bien voulu m’accorder.

Je remercie mes étudiants, ceux qui ont contribués à certains développements relatifs à ces travaux lors de projets internes de DUT informatique et les autres.

En dernier, je remercie ma famille et particulièrement mon père pour m’avoir toujours soutenu et encouragé dans ce travail.

Michel Dubois

Plan

Table des figures

Table des tableaux

Introduction

Lorsque l'on observe un groupe de robots évoluant simultanément dans un même environnement, il est difficile d'exprimer ce qu'il s'y développe, c'est-à-dire le fonctionnement interne des robots, les algorithmes au niveau individuels et collectifs.
Ainsi dans une compétition de robot comme celle de la RoboCup [Kitano & all, 97] (Figure 1), on voit s'affronter sur un terrain deux équipes de robots footballeurs. La compétition permet d'étudier les principales problématiques de la robotique collective (notamment le fait que les actions et les perceptions sont locales aux robots, l'absence de contrôle central, la nécessité de coopération) tout en conservant un caractère "réaliste" (imperfection de la perception et de la communication, incomplétude de l'information, hiatus entre intention (prévision) et action… possibilités de pannes). De fait, la RoboCup est considérée comme un benchmark de la robotique collective qui repose sur l'hypothèse du temps fini (la durée du match) et d'environnement fermé. La grande nouveauté consiste en la présence d'une équipe adverse. De plus ce benchmark propose des ligues où l'environnement et les règles du jeu sont adaptés afin d'embrasser un maximum de situations (ligues Small-size, Middle-size, standard à quatre pattes ou bipèdes, ligue de simulation).
On peut dresser les problèmes à surmonter tant au niveau individuel, que inter-individuel et au niveau collectif. Au niveau individuel, un robot doit se localiser sur le terrain, il doit reconnaître son contexte, il doit procéder à sa navigation par un calcul de trajectoires. Au niveau inter-individuel, les problèmes de coordination avec un partenaire, de calcul des passes, de choix d'une tactique se posent. Au niveau équipe, nous pouvons citer les problèmes de reconstruction du contexte, de mise en place d'une organisation, de définition de stratégies. Le but collectif d'une équipe peut s'énoncer ainsi : marquer le plus de buts, en encaisser le moins possible. Il doit se traduire en des sous-buts individuels tels que contrôler la balle, assurer la défense d'une zone, marquer un adversaire, etc. La RoboCup permet d'expérimenter plusieurs organisations expliquant comment "distribuer" les buts collectifs de façon adéquate. Dans une organisation statique, chaque joueur se voit confier un rôle fixe. Dans une organisation dynamique, chaque joueur prend un rôle selon le contexte dans lequel il se trouve. Dans une auto-organisation, il n'y a pas de rôles différenciés : les joueurs ont tous le même modèle de comportement.
Les équipes de la RoboCup ont fait le choix des approches délibératives ou réactives pour leur système multi robots (MRS) principalement en fonction des capacités possibles au niveau des compétitions. La vision globale autorisée dans la ligue Small-Size a conduit à des approches centralisées et délibératives. Dans la ligue Middle-Size où les robots sont pleinement autonomes, les approches réactives et délibératives sont présentes même si ce sont les dernières qui semblent préférées. Pour la ligue Sony Legged, on rencontre que des approches réactives du fait de la difficulté de communication entre robots.
Les points de vue que peuvent avoir l'observateur et le développeur sur la compétition sont différents.
Le plus évident est de décrire les règles du jeu : le contexte borné dans le temps, l'espace bi dimensionnel, les notions de distances, de compteurs, du nombre de joueurs.
Il est facile de décrire le partie physique du robot : nombre d'actionneurs, de capteurs, type de calculateur, type de structure mécanique … Il est également possible de définir les capacités de chaque robot en décrivant la liste des actions primitives qu'il est capable de produire.
Par contre décrire le comportement d'un robot au sein de l'équipe de robots à un instant donné, devient beaucoup plus délicat car il faut parler de communication, de coopération, de stratégie, de rôles, de contraintes temporelles … Bien entendu, il est encore plus compliqué de décrire l'activité simultanée de plusieurs équipes et a fortiori si chacune d'elle a un «coach » sur le bord du terrain qui donne des directives, le tout intégrant le jugement des arbitres dont on doit aussi décrire les comportements.
Un observateur expert, si on lui donne les règles du jeu, peut évaluer la stratégie d'une équipe, *at run-time*. Mais il a du mal à comprendre l'heuristique qui rapproche de l'accomplissement de la mission. *At compile-time*, le développeur peut traduire les règles du jeu en une fonction qui mesure la distance au but. Il doit manipuler des concepts de bas niveau comme l'envoi de messages, des méthodes, la manipulation de mémoires partagées, etc.
Un des challenge, c'est de combler le vide entre une perception intuitive et non formelle du jeu et son expression calculable et prouvable.

Dans un autre contexte, comme celui des composants robotiques auto reconfigurables, le caractère dynamique de l'organisation n'est pas une option mais une obligation. C'est notamment le cas pour les composants robotiques MAAM [Duhaut, 02] (Molecule = Atom | Atom + Molecule). Par exemple, dans une structure cyclique comme celle décrite dans la Figure 2, chaque composant s'y trouve dans une position relative. Pour que la chenille « roule », certaines pattes doivent aller au contact du sol, d'autres éventuellement doivent permettre de maintenir l'équilibre, d'autres doivent éviter les frottements qui peuvent ralentir la structure. Ainsi le comportement des

atomes au sol est différent de celui des atomes qui vont entrer au contact du sol et de celui de ceux en l'air. On peut dire que leur rôle respectif est différenciable. Dans un contexte auto reconfigurable, les composants changent de rôles. Dans la structure de typologie cyclique, un atome va jouer successivement le rôle « en l'air », le rôle « vers le sol », le rôle « au sol », le rôle « après le sol », etc. Faciliter cette succession dynamique de rôle est un challenge prioritaire de la robotique auto reconfigurable.

Figure 1 RoboCup standard

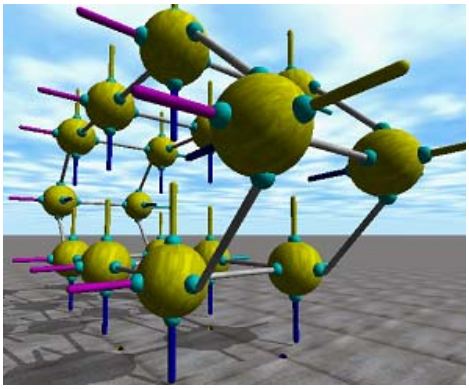


Figure 2 Une topologie MAAM cyclique

Dans ce mémoire nous proposons de définir un formalisme textuel MASL (Multi-Agents System Language) qui permet de décrire ce type d'activités des groupes de robots. Comme nous le décrirons par la suite ce formalisme s'apparente à un langage de programmation. Toutefois, il n'est pas directement exécutable sur une machine puisqu'il ne possède pas de compilateur ou d'interpréteur. Son implémentation n'a de sens qu'en utilisant un algorithme de réécriture propre à chaque robot afin de générer un code compatible avec le type de plateforme d'exécution embarqué sur le robot.

Le langage MASL tient compte d'un certain nombre de caractéristiques :

- il doit permettre une description «compacte » de l'activité d'un ensemble de robots comme celle d'une équipe de robots;
- les robots peuvent être différents dans leurs structures, dans leurs programmations, dans leurs plateformes d'exécutions embarquées. Par exemple dans la ligue Small-Size de la RoboCup, les équipes peuvent être constituées de petits robots à deux roues (DiffRobots) et des robots à trois roues (OmniRobots). Plutôt que de traiter ces robots de manière indifférenciée (les DiffRobots ignorent la consigne de la troisième roue), MASL doit pouvoir permettre l'utilisation optimale des deux types de robots au sein d'une équipe ;
- on doit avoir la possibilité de décrire la dynamique d'exécution multi-robot;
- les robots peuvent être soit indépendant, soit associés à une équipe de robots. Toutefois, durant leur exécution, les robots doivent pouvoir quitter une équipe pour aller en rejoindre une autre ;
- un robot n'est pas une entité parfaite, elle peut enrichir ou dégrader ses capacités durant son exécution donc dispose de différents modes d'exécution ;
- le nombre de robots peut être aussi grand que voulu.

Ces contraintes nous amènent à proposer la construction du langage MASL autour de six grandes propriétés :

- permettre de prendre en compte l'hétérogénéité des agents ;
- posséder des modes d'exécution pour rythmer l'activité d'un groupe au sein d'une équipe ;
- proposer différentes formes de communication entre robots ;
- introduire des « entrées dans le code » qui rendent simple les changements de comportement ;
- offrir un dispositif dit de « perméabilité » qui permet de modifier l'exécution des robots en prenant en compte la capacité actuelle de l'agent (avec la mise en œuvre de différents modes d'exécution et notamment la définition de modes d'exécution dégradés) et le caractère prioritaire pour lui de la collaboration en fonction des autres agents. La permeabilité garantit l'autonomie des agents.

- Permettre l'extensibilité : un programme pour un groupe de robot doit pouvoir s'exécuter pour un plus grand nombre de robots sans perte brutale de performance. Notamment il faudrait que les algorithmes puissent être exprimés de manière indépendante du nombre d'agents.

Malgré l'objectif robotique de ce travail et comme nous n'utiliserons des robots qu'à travers leurs primitives programmées d'action, de perception et de communication, le robot sera vu comme un agent logiciel. Bien que nous aillions voulu faire la différence entre le robot qui interagit effectivement sur l'environnement et l'agent qui le décrit informatiquement, nous pourrons par la suite confondre ces deux notions lorsqu'elle ne pose pas de d'ambiguïté, et c'est pourquoi MASL signifie multi-agent system language, il aurait été tout aussi juste de le nommer MRL multi-robot language.

Ce mémoire est composé de 3 chapitres, d'une introduction et d'une conclusion avec bibliographie et annexes.

Le chapitre 1 présente l'état de l'art en matière de programmation de robots et de programmation d'agents. Nous dressons la liste de contraintes que MASL doit pouvoir satisfaire

Le chapitre 2 est une introduction à MASL par l'exemple. Les notions fondamentales du langage y sont présentées ainsi que son expressivité.

Le chapitre 3 est une description de la syntaxe concrète et de la sémantique informelle de MASL.

L'annexe présente des résultats sur une implantation centralisée de MASL adaptée pour faire de la simulation notamment dans le contexte de projet Robea CNRS : Maam.

Chapitre 1 : Etat de l'art

Notre travail s'inscrit dans le domaine de la robotique collective. L'objectif est la programmation et le contrôle de groupe de robots afin de réaliser une tâche donnée. Du point de vue de cette programmation l'utilisation d'une approche multi-agents est pertinente.
En effet, outre des algorithmes spécifiques (planification tâche, génération de trajectoire …), le contexte robotique nécessite de prendre en compte l'exécution parallèle de plusieurs processus, leur synchronisation, les aspects temps réel, des capacités à interagir avec un environnement incertain, la tolérance aux pannes et les ressources limitées (énergie, mémoire, calcul) de l'embarqué.
Les systèmes multi-agents ont naturellement émergé d'une part car ils permettent d'exprimer les problèmes sous la forme de composants parallèles qui coopèrent à une même mission et par ailleurs, les composants peuvent exploiter des ressources distantes, distribuées sur un réseau.

Les systèmes multi-agents font partie de l'IAD (Intelligence Artificielle Distribuée) qui s'intéresse à la conception d'agents artificiels capables de s'organiser efficacement pour accomplir collectivement les fonctionnalités qui leur sont demandées. Outre les MAS (Multi-Agent Systems), les autres problèmes de prédilection de l'IAD sont le DPS (Distributed Problem Solving) et la PAI (Parallel Artificial Intelligence). Les recherches qui sont menées dans le domaine des MAS mettent moins l'accent sur le comportement individuel des agents que sur leurs capacités d'organisation et de fonctionnement collectif et portent sur les mécanismes de communication, de coopération, de coordination ou de négociation pouvant conduire des agents artificiels à accomplir ensemble les tâches qui leur seront confiées, en mobilisant aussi bien leurs facultés individuelles que les ressources de l'organisation qu'ils forment.

Les agents sont situés dans un environnement physique ou social, c'est-à-dire dotés d'une enveloppe corporelle[1] où les contraintes physiques de l'environnement s'exercent fortement et d'une socialité minimale, lorsque l'on a comme objectif qu'ils manifestent ou assimilent des comportements intelligents (individuels comme collectifs)

Dans ce chapitre nous allons étudier la programmation des MAS, mais au préalable nous étudierons aussi différents travaux sur les calculs parallèles pour que notre langage puisse effectivement avoir une implémentation réaliste sur différents types d'exécution. L'objectif robotique de notre travail nous amènera également à étudier les différents modes de programmation de robot ainsi que des architectures logicielles qui les organisent. Enfin, nous identifions les faiblesses et les caractéristiques originales qui peuvent être concentrée au sein d'un langage dédié au contrôle d'agents robotiques multiples. La présentation va aborder les théories sur lesquelles sont bâties les architectures de contrôle d'agent, de systèmes multi-agents et de systèmes dédiés robotiques.

1.1 Préalables

Une architecture robotique est un système complexe faisant intervenir un grand nombre de capteurs et d'effecteurs et assemble les composants logiciels et matériels utilisés dans un robot et définit les modalités d'interaction de ces composants. Une architecture logicielle se restreint à l'aspect logiciel. Elle propose une méthode de définition et d'organisation des différents composants. Elle précise le nombre de couches, les fonctions satisfaites à leur niveau, les contraintes temporelles qui leur sont liées. Les couches sont autant de boites noires dont on connaît les entrées et les sorties. Pour répondre aux fonctionnalités, plusieurs composants peuvent être candidats. Lorsqu'elle répond à la problématique d'évolution, elle permet d'ajouter à froid ou à chaud de nouvelles fonctionnalités sans remise en cause de l'existant. Elle permet la prise en compte de la sécurité par des actions de recouvrement (en cas d'échec) et de compensation. Des fonctionnalités décisionnelles sont souvent nécessaires pour rendre les robots autonomes.

[1] Le courant actuel du « network sensor » vise à rendre cette enveloppe corporelle moins fondamentale. Via l'accès au réseau, un robot intègre une multitude de capteurs qui ne sont pas physiquement liés à lui et qu'il partage avec d'autres robots.

La notion de comportement est la brique fondamentale de l'intelligence naturelle. Un comportement est la connexion (mapping) entre des entrées capteurs et un modèle d'actions moteurs qui sont ensuite utilisées pour réaliser la tâche. Des comportements simples opérant indépendamment dans des agents peuvent conduire le groupe à une séquence d'actions complexes (pour un observateur extérieur).

Niveaux de description d'un agent

Lorsque nous définirons les systèmes multi-agents nous parlerons de modèle d'agents, d'architecture d'agents et d'implémentation d'agents. Il est donc nécessaire d'expliciter ces termes qui constituent les différents niveaux de description d'un agent qui sont [Wooldridge & all, 95] :
- Le modèle qui décrit comment l'agent est compris, ses propriétés et comment on peut les représenter.
- L'architecture qui est un niveau intermédiaire entre le modèle et le contrôle et l'implémentation. Elle précise la création du système c'est-à-dire les propriétés qu'il doit posséder conformément au modèle et les liaisons avec les autres agents.
- L'implémentation qui s'occupe de la réalisation pratique de l'architecture des agents à l'aide de langages de programmation.

Un agent est une entité logicielle qui fonctionne continuellement d'une manière autonome dans un environnement. Il est basé sur le modèle réactif (il interagit avec son environnement via des capteurs et des actionneurs) ou sur le modèle proactif (il raisonne à partir de sa propre représentation du monde). Lorsque ce dernier contient d'autres agents, on parle de SMA (Système Multi-Agents) : le caractère social des agents recouvre leur aptitude à communiquer, coopérer et collaborer à une mission commune.

Modèle de programmation

Un modèle de programmation définit l'ensemble des concepts (structures de données, instructions permettant de les manipuler) ainsi que leur sémantique qui permet notamment d'évaluer le pouvoir d'expression du langage. Le modèle de programmation est le niveau d'abstraction fourni par un environnement de programmation au programmeur d'application.

Modèle d'exécution

Un modèle d'exécution est l'ensemble des mécanismes qui permettent de mettre en œuvre l'ensemble des concepts définis par le langage de contrôle, dans une machine d'exécution. Il peut exister pour un modèle de programmation, plusieurs modèles d'exécution. Pour un ordinateur ayant un modèle d'exécution de Von Neumann (la mémoire contient le programme et les données d'entrée et de sortie, le processeur lit une instruction en mémoire, la décode et l'exécute de façon consécutive puis l'opération est répétée), les langages de programmation offrent différents modèles de programmation.

Cette distinction entre modèle de programmation et modèle d'exécution peut être résumée selon Luc Bougé [Bougé, 96] par : Calcul Abstrait = Modèle d'Exécution + Modèle de Programmation.

Le programmeur écrit son programme dans un langage qui offre un modèle de programmation et il fera appel à un traducteur qui transcrit le programme dans le langage du support d'exécution qui suit un modèle d'exécution. Le cas le plus courant de traducteur est celui du compilateur qui traduit le code d'un langage de programmation en du code machine, éventuellement celui d'une machine virtuelle (cas de java). Plus le traducteur est évolué, plus le modèle de programmation s'éloigne du modèle d'exécution.

Modèle de programmation parallèle

Un modèle de programmation parallèle doit permettre au programmeur d'exprimer la sémantique parallèle du programme, c'est-à-dire :
- la concurrence : le fait que des activités s'exécutent en parallèle ;
- la synchronisation : la coordination des activités concurrentes ;
- la distribution : la répartition des données et des calculs entrant en jeu dans les activités concurrentes,
- les communications : les échanges de données entre les activités concurrentes.

Ces caractéristiques du parallélisme sont liées entre elles.

1.2 Concurrence et distribution

Dans cette partie nous présentons les méthodes traditionnellement retenues pour la programmation de systèmes ayant de la concurrence et de la distribution. Nous verrons que le langage MASL proposé s'inspire pour partie de ces méthodes.

Flynn [Flynn, 72] propose une classification des modèles d'exécution parallèle en 4 catégories. Ils diffèrent selon le nombre représenté par S (Single) ou M (Multiple) du point de vue du flot d'instruction I (ce que fait l'ordinateur) ou du flot de données D (sur quoi il doit le faire) qui sont actif à chaque point de calcul :

- SISD (instruction simple, une seule mémoire) - un ordinateur séquentiel qui exploite aucun parallélisme autant au niveau des instructions que de la mémoire. Cette catégorie correspond à l'architecture de von Neumann et supporte le parallélisme apparent ou pseudo parallélisme où des processus concurrents se partagent l'unique processeur.
- SIMD (instruction simple, plusieurs mémoires) - Single Instruction Multiple Data - un ordinateur qui utilise le parallélisme au niveau de la mémoire, par exemple le Processeur vectoriel qui comprend un vecteur de couples (processeur, mémoire).
- MISD (instruction multiples, une seule mémoire) - Multiple Instruction Simple Data - Une même donnée est traitée par plusieurs processeurs en parallèle. Il y a peu d'implémentation en pratique, peut-être utilisé dans le filtrage numérique et la vérification de redondance dans les systèmes critique.
- MIMD (instruction multiples, plusieurs mémoires) Multiple Instructions on Multiple Data - Plusieurs processeurs traitent des données différentes car chaque processeur possède une mémoire distincte. C'est l'architecture parallèle la plus utilisée. Il y a deux variantes soit :
  1. MIMD à mémoire partagée - Les processeurs ont accès à la mémoire comme un espace d'adressage global. Tout changement dans une case mémoire est vu par les autres CPU. La communication inter-CPU est effectuée via la mémoire globale.
  2. MIMD à mémoire distribuée - Chaque CPU a sa propre mémoire et système d'exploitation. Nécessite un middleware pour la synchronisation et la communication

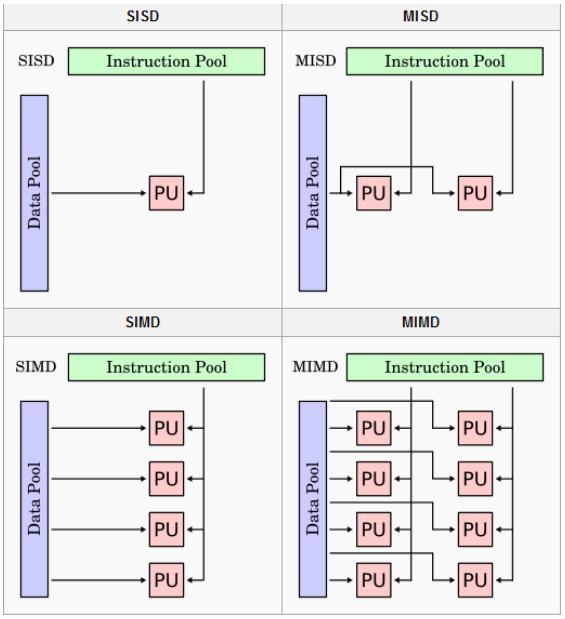


Figure 3 Les différents modèles d'exécution parallèle

Le modèle d'exécution PRAM

Le modèle d'exécution PRAM [Fortune & all, 78] utilise implicitement une opération de synchronisation globale et explicitement une structure de mémoire partagée. Ce qui se fait en une étape synchrone sur la PRAM nécessite un protocole complet sur la MIMD. Il comprend l'ensemble des actions suffisantes pour réaliser une étape globale de lecture/écriture PRAM : synchronisation globale et communication point-à-point.

Le modèle d'exécution BSP

Le modèle BSP introduit les barrières de synchronisation entres processus qui peuvent être utilisé dans le run-time MASL pour effectuer les synchronisations entre agents (*Cf.* chapitre 3).

Le modèle d'exécution BSP [Valiant, 90] (Bulk Synchronous Parallelism) représente le calcul parallèle sur p processeurs identiques comme une succession de super-étapes : alternance de calculs séquentiels (p calculs asynchrones) et de communications (échanges de données entre les processeurs à l'aide d'une synchronisation globale) d'où l'adjectif bulk-synchronous.
Un programme parallèle suivant le modèle BSP est une suite séquentielle de méta-étapes (supersteps) qui occupent conceptuellement les p processeurs de la machine parallèle sur laquelle le programme BSP s'exécute. Une méta-étape se décompose en 3 phases ordonnées :
1. une phase de calculs locaux n'utilisant que les mémoires locales à chacun des processeurs,
2. une phase de communication, impliquant des transferts de données entre les (p) processeurs,
3. une phase de synchronisation globale (*barrier synchronization*) qui attend que tous les transferts de données soient terminés.

Le modèle de coût BSP donne une estimation du temps d'exécution en associant le temps séquentiel des opérations, la rapidité séquentielle des processeurs, le temps nécessaire à une 1-relation (phase de communication où chaque processeur envoie ou reçoit au plus un mot) `g` (qu'on appellera pour simplifier temps de communication) et la latence `L` du réseau par une simple formule basée sur p.
Le temps d'exécution d'une super-étape se définit par le temps pris par le calcul séquentiel suivit d'une synchronisation et est estimée par :
$\text{Temps} = \max(w_i \text{ pour } 0 \le i < p) + \max(h_i \text{ pour } 0 \le i < p) \times g + L$
où $h_i = \max(h_{i+}, h_{i-})$ et h_{i+} (resp. h_{i-}) est la taille totale des messages envoyés (resp. reçus) par le processeur `i` durant la super-étape de communication. Ici, w_i est le temps séquentiel du processeur `i` durant la super-étape de calcul.
L'estimation a été démontrée fine pour la plupart des situations du calcul parallèle. Les exceptions sont dues aux calculs ayant des messages de petite taille et beaucoup de barrières, et aux architectures hétérogènes. Malgré ces exceptions, BSP est un modèle d'architecture parallèle simplifié et portable. Il assure l'absence d'interblocage et le déterminisme des calculs, en plus de fournir un modèle d'évaluation des performances parallèles.

La communauté étudiant le domaine des calculs à hautes performances s'est intéressée à des réseaux particuliers de machines, appelés clusters. Construire un cluster consiste à assembler un ensemble important de machines sur un réseau local à haut débit. On peut considérer cela comme une version économique des anciennes machines massivement multi-processeurs, qui est rendu possible grâce à l'accroissement des débits des réseaux. Sur ce genre d'environnement matériel, des langages se focalisant sur l'échange de messages sont apparus comme PVM [PVM, 81], [Geist & all, 94] (Parallel Virtual Machine) ou MPI [MPI, 89], [Hempel, 94] (Message Passing Interface). Ces langages sont un premier pas important concernant l'indépendance du langage de programmation vis-à-vis de l'environnement d'exécution. Cependant, cette indépendance reste relative et nécessite la compilation du programme sur chacune des architectures hôtes.

1.2.1 Langages data-parallèles

Dans le parallélisme de contrôle, le programme décrit explicitement le parallélisme du point de vue calcul (création, terminaison, communication, synchronisation). Cela aboutit à des programmes difficiles à écrire, pas prouvables en absence de l'état global, non déterministes et soumis au phénomène d'inter blocages (*deadlocks*).
Dans le parallélisme de données, des instructions définies sur des ensembles de données permettent que le même traitement soit réalisé sur chacune des données de l'ensemble. Les instructions data-parallèles se traduisent par des traitements élémentaires sur les données.

Le programme, dans son ensemble, est composé d'une séquence d'instructions globales sur des collections de données. La programmation séquentielle étant cognitivement relativement aisée, cet aspect de la programmation par parallélisme de données consiste en fait à se rapporter à un mode de programmation connu et déterministe. Lorsqu'une même instruction est appliquée à tous les éléments d'une collection de données, ces applications peuvent être effectuées en parallèle. Le programmeur exprime ses algorithmes comme des opérations globales sur des collections de données et non sur chaque élément de la collection. Il n'est pas nécessaire au programmeur de penser à chaque élément de la collection individuellement.

Dans la pratique, les langages mixent le modèle data-parallèle avec le modèle scalaire. On peut manipuler des données scalaires avec des opérateurs scalaires et des données vectorielles avec des opérateurs vectoriels. De

plus certains opérateurs permettent de passer du monde scalaire au monde vectoriel (diffusion ou broadcast) ou l'inverse (concentration ou fold)

Le principe de la séquence d'instructions parallèles a été exposé par Luc Bougé dans [Bougé, 96].
Etant donné les processus P_1, P_2, Q_1 et Q_2, alors les modèles de calculs parallèles possibles sont :
- une composition parallèle de programmes séquentiels (ou PARofSEQ) notée $(P_1;P_2)||(Q_1;Q_2)$;
- une composition séquentielle de programmes parallèles (ou SEQofPAR) notée $(P_1||Q_1);(P_2||Q_2)$.

Il pose le contraste entre le PARofSEQ du parallélisme de tâches et le SEQofPAR du parallélisme de données.

Le modèle d'exécution PARofSEQ est traduisible depuis un modèle de programmation SEQofPAR. Le compilateur passe d'un modèle où le code est exprimé de manière synchrone et centralisée à un modèle d'exécution éventuellement asynchrone et distribué. On passe d'un modèle plus contraint vers un modèle moins contraint et on démontre une équivalence de programme.

La construction synchrone FOR ALL ne peut pas être exécutée directement sur les machines MIMD. On traduit les FORALL synchrones en des séquences équivalentes de FORALL asynchrones. Dans l'exemple suivant, on donne la transformation source à source.
Les transformations de code ont pour objet d'adapter les programmes à l'architecture cible, elles sont effectuées soit au niveau du code source, soit au niveau du code machine. Quand un programme est transformé, la sémantique de ce programme doit rester identique. Une transformation est dite légale si pour tout programme correct (respectant la sémantique du langage), le programme transformé et le programme original produisent exactement les mêmes résultats. Un exemple de transformation légale qui modifie le comportement d'un code est montré en Figure 4 à partir des mêmes entrées.

La transformation montre de manière explicite les barrières de synchronisations ainsi que les variables temporaires `H1` et `H2`. Ces dernières sont nécessaires afin de ne pas écraser des valeurs avant qu'elles ne soient lues. Une barrière de synchronisation est un mécanisme de coordination de plusieurs threads concurrents. Quand un thread attend à une barrière, son exécution est suspendue jusqu'à ce que tous les threads participant à la barrière attendent eux aussi à la barrière. Après ce rendez-vous, l'exécution de chaque thread suspendu reprend. Dans plusieurs modèles data-parallèles, les processeurs sélectionnées s'engagent dans un rendez-vous implicite après l'exécution d'une opération data parallèle. La séquence de FORALL asynchrone code explicitement de tels rendez-vous. Notons que si le programme transformé est équivalent, il est très synchrone. Eventuellement certaines barrières de synchronisations ne sont pas nécessaires pour conserver la sémantique.

```
FORALL i : [1..N] IN SEQ
      A[i] := A[i+1];
      B[i] := B[i+1];
END
```

≡

```
FORALL I : [1..N] IN PAR
      H1[i] := A[i+1];
END; //Synchronisation
FORALL i : [1..N] IN PAR
      A[i] := H1[i];
END; //Synchronisation
FORALL i : [1..N] IN PAR
      H2[i] := B[i+1];
END; //Synchronisation
FORALL i : [1..N] IN PAR
      B[i] := H2[i];
END; //Synchronisation
```

Figure 4 Transformation de programmes valide

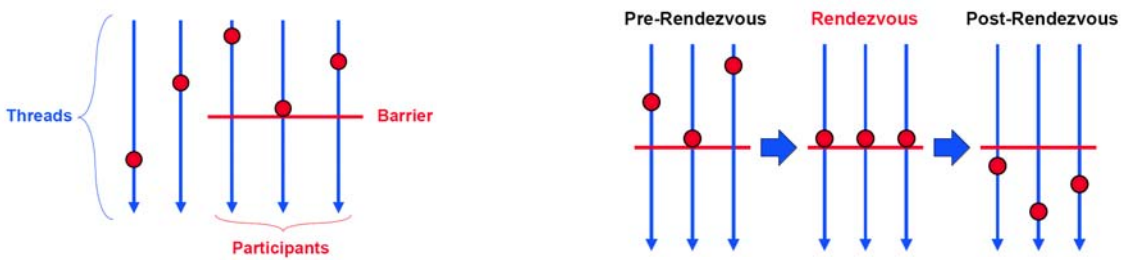


Figure 5 Principe des barrières de synchronisation

La construction `where B do S`, ou `B` est une expression vectorielle logique évaluée localement par chaque composant, permet d'obtenir un asynchronisme limité. Tous les composants pour lesquels l'expression `B` est évaluée à `true` exécutent le programme `S` de manière synchrone tandis que les autres ne font rien. Cette construction introduit de l'asynchronisme en masquant les composants ne vérifiant pas une condition locale et donc en généralisant le `IF` scalaire.

La gestion de la distribution implique un certain nombre de calculs élémentaires mais répétitifs, tels que ceux visant à déterminer l'identité du nœud propriétaire d'un élément (i; j) donné et son adresse locale sur ce nœud.
On procède à la parallélisation en redéfinissant certains opérateurs de telle sorte que, pendant les calculs, chaque nœud ne s'intéresse qu'aux données dont il est propriétaire. Ce type d'exécution est conforme à la règle des calculs locaux (owner compute rule). Les techniques de parallélisation mises en œuvre sont les mêmes que celles que l'on tente de faire appliquer automatiquement par les compilateurs-paralléliseurs de type HPF [COE, 95].

Le modèle implicite de la programmation par parallélisme de données est celui d'une machine à mémoire distribuée, par conséquent avec des coûts de communications de données entre processeurs. Le bon placement des données minimise le nombre ou la distance de ces communications.

Les langages data parallèles objets proposent un modèle abstrait de machine parallèle. Les objets et les threads sont placés dans ce modèle au moyen d'un nombre abstrait de processeurs au lieu d'être placé directement. Le modèle est ensuite automatiquement placé au niveau de la topologie concrète.

Les langages data parallèles objets (HPJava [Carpenter & all, 98]) proposent eux aussi comme modèle abstrait de machine parallèle, celui de la grille multi-dimensionnelle et la correspondance des tableaux se fait par rapport à cette grille. La localité d'un objet peut s'exprimer en retenant pour chaque tableau le même indice. Le modèle HPJava est explicitement SPMD. La construction `on` restreint l'exécution d'une portion de code aux processeurs qui sont élément d'un groupe particulier. L'abstraction `location` est un paramètre de la construction `at` qui autorise l'accès à un élément de la grille qu'à certains processeurs.

Huit et demi [Michel, 96] (prononcer « otto et mezzo ») est un modèle de calcul parallèle pour les grandes simulations des systèmes dynamiques. Il se concrétise par un langage flot de données permettant de manipuler des collections et d'exprimer à la fois du parallélisme de contrôle et du parallélisme de données. Un programme correspond à un système d'équations à résoudre dans l'ensemble des flux de collections (suite infinie de valeurs). Son exécution correspond à calculer séquentiellement, pour chaque indice du flux, les collections qui résolvent le système. Pour construire plus facilement des simulations, on peut combiner des streams qui ont des horloges différentes par une logique d'observation d'état, contrairement aux langages synchrones. MGS [Giavitto & all, 01] (Modèle Général de Simulation) est un langage déclaratif fonctionnel dédié à la simulation de systèmes dynamiques en particulier dans le domaine biologique. Il s'inscrit dans la continuation de 8½. Il n'est pas purement flots de données mais des simulations se présentent comme des flots de collections. Les collections sont topologiques (un ensemble d'éléments avec une relation de voisinage). MGS repose sur les transformations de ces collections. Dans le parallélisme de réduction (cas des langages fonctionnels et ceux de flots de données), une instruction est sélectionnée et exécutée lorsque ses opérandes sont prêtes ou calculées. Il n'y a pas d'expression possible du séquencement par le programmeur. Il est construit automatiquement à partir des dépendances entre les données du programme. MGS est à rapprocher des langages de systèmes hybrides (cf §1.6) tels que FROB [Peterson, 99] (Functional ROBotics), CHARON [Alur & all, 00] (Coordinated control, Hierarchical design, Analysis, and Run-time mONitoring of hybrid systems) et CCL [Klavins, 03], [Waydo & all, 03] (Computation and Control Language) (*Cf.* 1.6.2) qui sont prévus pour raisonner sur les stratégies de contrôles multi-robots.

Ainsi le modèle data parallèle est né en réaction à la trop grande difficulté à écrire des programmes suivant le parallélisme de contrôle dans le cas ou le nombre de tâches est très grand. La problématique de l'exploitation du parallélisme de données par un langage data-parallèle avant 1990, s'exprimait en termes «parallélisme de gain fin, SIMD ou vectoriel, mapping ». Ainsi, l'importance de la notion de minimisation des communications dans le parallélisme de données est telle que la notion de placement de données a pu être considérée comme l'essence du parallélisme de données. Après 2000, l'exploitation du parallélisme de données par un langage data-parallèle utilise les termes « gros gain, MIMD, loop-partitionning ». L'aisance de la programmation data parallèle avec la mise en parallèle de manière implicite sont des apports précieux pour MASL.

1.2.2 Langages d'acteurs

Les langages d'acteurs sont intéressants car ils introduisent une autonomie des agents dans la mesure où un agent est un processus. Il offre la communication par envoie de messages asynchrones et aussi une interface dynamique pour les objets actifs. Nous retiendrons ces idées dans MASL.

Les langages d'acteurs sont une sous-famille des langages à objets adaptée au parallélisme. Les objets sont actifs, autonomes et concurrents, et ainsi appelés acteurs. Leur autonomie vient de l'unification des concepts de processus et d'objets passifs. Un objet actif est un objet ayant sa propre activité privée. Un objet qui n'est pas actif est passif. A la différence des objets passifs qui communiquent par transmissions de messages synchrones, le mode primitif de communication entre acteurs est asynchrone, ce qui leur évite d'attendre inutilement la fin et le retour de valeurs des requêtes de communication.
Dans le paradigme des variables partagées, les processus communiquent en écrivant et en lisant à partir d'un endroit de la mémoire partagés. La gestion prend en compte les phénomènes d'exclusion mutuelle (un seul processus peut accéder à un ensemble de variables partagées) et la synchronisation conditionnelle qui oblige que des conditions préalables soient vérifiées pour permettre l'exécution d'un programme, d'un processus et des sections critiques. Plusieurs processus exécutent alternativement des parties de programmes où ils accèdent à des variables partagées et d'autres parties sans accès à ces variables. Les sections critiques sont les parties des programmes où il y a des accès à des variables partagées. Le problème est d'obtenir l'exclusion mutuelle en évitant les interblocages et des délais trop grands et que chaque processus candidats à l'entrée de la section peut espérer y entrer. Des solutions à ce problème sont les sémaphores, les moniteurs ou les régions critiques conditionnelles.
Le modèle des acteurs s'abstrait de ces problèmes de synchronisation en encapsulant l'état d'un objet et son thread d'exécution et en limitant les communications à l'envoi asynchrone de messages.

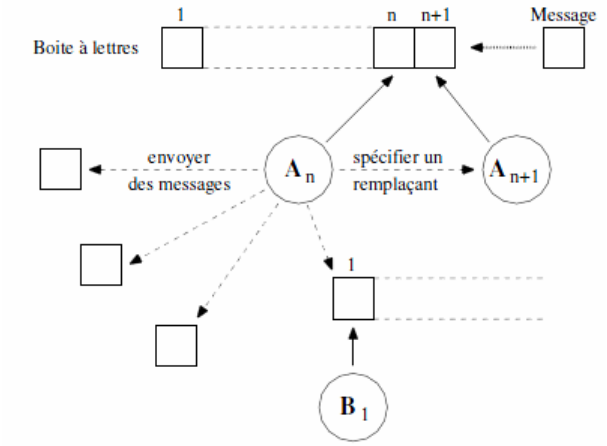


Figure 6 Comportement d'un acteur lorsqu'il traite le nième message de sa boite aux Lettres.

Chaque acteur dispose d'une boite aux lettres à partir de laquelle il traite successivement les messages qu'il a reçus. Chaque message est associé à un script qui peut envoyer des messages à d'autres acteurs (ses accointances) ou créer de nouveaux acteurs. Une fois le script terminé, il spécifie un comportement de remplacement qui devra traiter les messages suivants.
Le concept d'acteur prend sa source dans les travaux de Carl Hewitt [Hewitt, 77]. Le langage ABCL/1 [Yonezawa & all, 87] (Actor Based Concurrent Language) nous présente une approche qui combine la programmation par acteurs avec la programmation plus traditionnelle : le programmeur définit des acteurs de grain moyen dont les comportements s'expriment en partie dans le langage de base. Un acteur ABCL/1 offre à l'utilisateur plusieurs protocoles de communication évolués : synchrone, anticipé, interruptions, en plus du seul mode de communication asynchrone primitif entre les acteurs. En faisant cohabiter ces différents modes de communication, le modèle d'acteur tel que définit initialement doit être adapté pour garantir une sémantique unique. Un objet passif devient actif en répondant à une requête. Un objet actif devient passif en terminant l'action associée avec une requête.
Un système à objets actif permet la concurrence inter-objet par l'indépendance des activités des objets. Dans le cas général, un objet actif ne peut traiter qu'une requête à un moment donné. Dans certains modèles de programmation, un objet actif peut traiter des requêtes de manière simultanée. Un tel objet possède plusieurs activités internes, ce qui aboutit à la concurrence intra-objet, difficilement implantable et nécessitant des

mécanismes pour la consistance des données. Pour des raisons d'efficience, les objets complètement concurrents sont généralement implantés comme objets passifs et ils sont répliqués sur chaque processeur.

La communication asynchrone (lorsque l'émetteur n'est pas bloqué dans l'attente de la lecture de son message par le destinataire) conduit à la notion de variables futures [Halstead, 85], introduite en premier dans MultiLISP, qui font de l'attente par nécessité. L'interruption (terminaison prématurée) de méthodes non bloquantes peut nécessiter l'appel implicite à une méthode qui garantit l'état cohérent de l'objet.

En outre, l'interface d'une classe est statique en programmation par objets et donc en UML, alors qu'elle est dynamique pour un acteur. Le modèle pur d'acteur [Agha, 86] combine le concept d'objet avec le concept d'interface dynamique. Quand une interface dynamique d'un acteur change, cet acteur devient un nouvel acteur avec un comportement différent, récupérant les anciennes valeurs des variables d'instance. Au lieu de changer dynamiquement si une méthode peut ou non être appelée, le code implémentant le comportement d'acteur est permuté dynamiquement par l'ordre `become`. Les commandes `include` et `exclude` permettent d'ajouter ou de retirer une méthode de l'interface dynamique. A noter que si une méthode est appelée alors qu'elle ne fait pas partie de l'interface actuelle, l'appel est mis en attente jusqu'à ce que cette méthode fasse à nouveau partie de l'interface dynamique. La programmation de cette interface dynamique est verbeuse dans la mesure où chaque méthode doit être incluse / exclue individuellement.

Le concept d'acteur est probablement la fondation la plus appropriée pour décrire et implanter des systèmes multi-agents [Ferber, 87]. Un objet actif étant base naturelle au concept d'agent autonome. On peut simuler un agent à l'aide d'un objet mais cela demande beaucoup de travail. Les primitives offertes par un langage d'agent doivent permettre de se focaliser sur la définition des comportements des agents.

1.2.3 Langages réactifs

1.2.3.1 Langages asynchrones

S'il est soumis à des contraintes temps réels, l'agent peut être considéré comme un système réactif dans lequel déterminisme, sureté de fonctionnement, maîtrise de la rapidité de la réaction sont alors recherchés. Dans l'approche asynchrone, notamment Electre [Roux & all, 92], SDL [SDL 2000] (Specification and Description Language), les occurrences d'événements peuvent être perçues continuellement. Chaque instant constitue un point qui peut être infiniment prêt d'un autre sans être confondu ou considéré comme simultané : les actions ont une durée non nulle. L'asynchronisme se traduit par une relativement importante indépendance des processus entre eux qui peuvent, de ce fait, être très naturellement distribués sur des sites différents, et évoluant à des rythmes différents, avec des synchronisations ponctuelles. L'interruptibilité des méthodes en cours d'exécution est une caractéristique des langages asynchrones. Ces langages n'utilisent pas un temps absolu comme l'instruction `delay` dans [ADA, 83] mais plutôt un temps logique divisé en instants qui correspondent aux moments de réaction des programmes.

1.2.3.2 Modèles synchrones

Nous allons faire successivement la distinction entre le modèle synchrone qui considère qu'un instant est de durée nulle, le modèle réactif synchrone défini par Boussinot qui considère des instants à durée non nulle et son extension distribuée.

Modèle synchrone

Dans l'approche synchrone ([Halbwachs 98], Esterel [Boussinot & all, 91], Lustre [Halbwachs & all, 91], Signal [Le Guernic & all, 91]) un instant est considéré de durée nulle. Dans un cycle d'horloge ou instant, plusieurs composants s'exécutent en parallèle. Tous les composants perçoivent les évènements ou signaux de la même manière, au cours des mêmes instants. Elle autorise une réaction instantanée à l'absence d'évènement. Un évènement généré est perçu dans le même instant. Les évènements peuvent être engendrés à tout moment par l'environnement. Ils sont éphémères : leur présence varie dans le temps. De plus ils peuvent être également utilisés « en interne », comme moyen de communication à l'intérieur du système lui-même. Ces langages transfèrent les traitements consommateurs de temps aux langages hôtes (C ou ADA). Cependant ils sont mal adaptés pour la programmation non réactive, en particulier pour les tâches délibératives (voir ci-dessous). Signal, Lustre et Lucid Synchrone [Caspi & all, 96] sont des langages flots de données qui manipulent des suites infinies de valeurs comme objets de base.

Modèle réactif synchrone

L'approche réactive synchrone considère des instants de durée non nulle. ReactiveC [Boussinot, 91] inclut des extensions au langage C pour la programmation réactive. Il est utilisé comme noyau pour l'implémentation de plusieurs formalismes réactifs car il permet de contrôler l'ordonnancement à l'intérieur d'un instant. Un objet réactif dans le modèle d'exécution ROM [Boussinot & all, 95] (Reactive Object Model) encapsule des données partagées par ses méthodes qui sont des instructions réactives. Une machine réactive exécute les instructions réactives des objets partageant le même instant. Un instant est terminé lorsque tous les objets ont fini de réagir à tous les évènements. Les objets réactifs réagissent instantanément à la présence des évènements, et de manière retardée à leur absence. Un objet ne peut pas supprimer un évènement. Par contre, les objets réactifs peuvent être créés ou détruits dynamiquement à tout moment de l'exécution. L'opération interne d'inhibition dans le modèle réactif peut ainsi être obtenue par la suppression de l'instance gérant la compétence à inhiber. La communication entre objets se fait par appels asynchrones de méthodes (appels non bloquant ou envois de consignes). Cette dernière sera exécutée dans l'instant ou sera rejetée. Si durant un même instant plusieurs objets appellent une même méthode, un seul objet verra sa consigne acceptée, toutes les autres consignes seront rejetées. Ainsi dans ce modèle, les évènements permettent de faire communiquer les objets réactifs au sein d'un seul agent.

Modèle réactif synchrone distribué

On peut considérer qu'un système multi-agents est tout simplement un ensemble d'agents partageant un environnement commun. L'environnement met en relation l'ensemble des agents : il est le lieu de leurs interactions. Les systèmes multi-agents robotiques prennent en charge la distribution et les moyens de communication possibles entre agents : envoi de message, mémoire virtuellement partagée et physiquement répliquée, broadcast d'évènements. Les Communicating Reactive Processes [Boniol, 95] sont des processus Esterel distribués communiquant par rendez-vous. Dans le modèle DROM [Boussinot & all, 98] (Distributed Reactive Object Model), chaque agent est doté d'une machine réactive dans laquelle se trouvent des instances d'objets réactifs. Les machines réactives sont réparties sur le réseau. Les machines réactives synchronisées forment une aire réactive. Plusieurs composants appartenant à une même aire réactive partagent le même instant et communiquent par broadcast à l'aide d'un composant appelé synchroniseur. A un moment donné, une machine ne peut être connectée qu'à un seul synchroniseur. La communication asynchrone se fait par des évènements inter-zone réactives. Ce modèle repousse l'asynchronisme de la frontière de l'agent à celle de l'aire réactive. Un système composé de machines réactives liée à des synchroniseurs peut être vu comme des aires réactives évoluant selon les connections et déconnexions dynamiques des machines réactive aux synchroniseurs. Le modèle DROM n'offre pas la notion de sous aire réactive : il n'est pas possible d'atteindre une sous partie. On se trouve en présence des GALS [Halbwachs & all, 02] (Globally Asynchronous Locally Synchronous) où chaque site est un système réactif déterministe et où les communications entre sites sont asynchrones.

Les approches synchrones sont aussi à l'origine de travaux qui ont influencé les architectures pour robot autonomes mais restent cantonnées aux couches de bas niveau de l'architecture. Elles sont peu adaptées à la mise en place de systèmes décisionnels. Ainsi, les exécutifs des robots embarquent des objets passifs, des objets actifs et des objets réactifs.

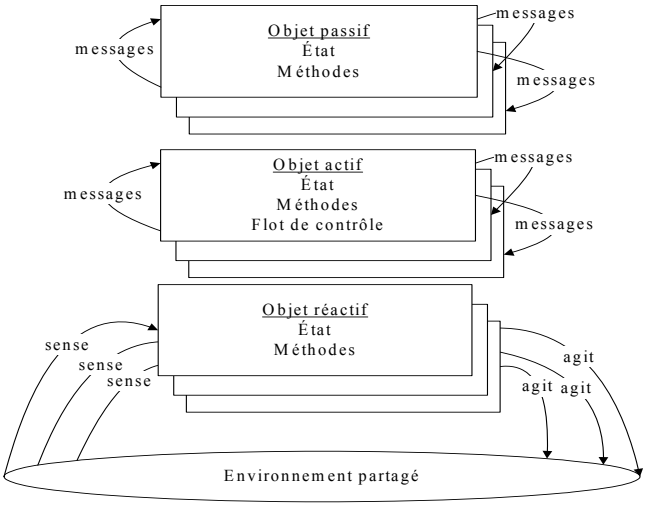


Figure 7 Objets passifs, objets actifs et objets réactifs.

1.3 Architectures d'agents robotiques

Les architectures d'agents robotiques se caractérisent par une mise en compétition des modules pour remplir la mission. Ces modules sont généralement programmés dans les langages introduits ci-dessus. Ces architectures doivent être implémentables dans tout langage spécifique à la robotique en conséquence MASL permettra de les mettre en oeuvre. Pour une description plus précise et comparée des principales architectures utilisées en robotique autonome, on peut voir [Ingrand, 05]. A noter que du point de vue du fonctionnement interne d'un agent, selon [Russel & all, 06], les agents réflexes simples répondent aux percepts, tandis que les agents réflexes fondés sur des modèles maintiennent un état interne afin de suivre l'évolution des aspects du monde non discernables dans le percept courant. Les agents fondés sur des buts agissent pour atteindre des objectifs tandis que les agents fondés sur l'utilité essaient de maximiser leur « satisfaction » espérée. La Figure 8 montre les trois types d'architectures robotiques logicielles.

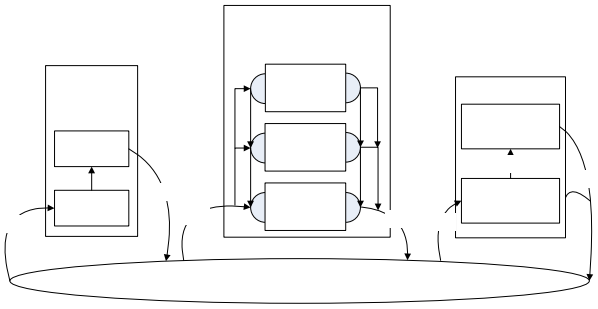


Figure 8 : Les agents mettent en œuvre des modèles d'exécution qui implémentent les concepts délibératifs, réactifs ou hybrides.

1.3.1 Agent délibératif

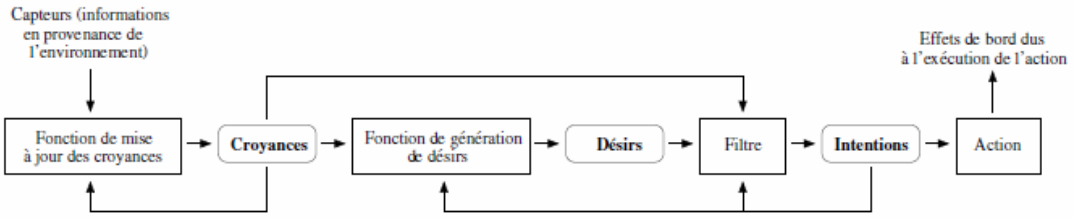


Figure 9 Une architecture schématique d'agent BDI

Du caractère autonome de l'agent est apparu le concept de but de cette entité intelligente dont les choix sont fondés sur des attitudes mentales suivant le modèle BDI [Bratman, 87] (Believes, Desires, Intensions) : ses croyances, ses désirs, ses intensions. Il doit mettre à jour ses croyances (son état) avec les informations perçues de son environnement, décider quelles alternatives lui sont offertes, les filtrer afin de déterminer de nouvelles intentions et réaliser ses actions selon ses intentions (théorie du passage à l'acte). Le concept d'intention permet de relier les buts aux croyances. Un agent délibératif contient donc une représentation symbolique du monde et choisi les actions à accomplir à l'aide d'un raisonnement logique.
Le moteur d'exécution d'un agent est décrit par un interpréteur contenant une exécution en boucle dont la description est présentée dans la figure ci-dessus (Figure 9). Au début du cycle, on exécute une sélection parmi les événements de la queue. Le générateur d'options lit les événements sélectionnés et retourne les meilleures options (alternatives de plans) qui sont déclenchées par ces événements. Dans l'étape de décision (appelée aussi délibération, on sélectionne les plans à exécuter. Pour des raisons d'efficacité, le processus de délibération fait appel à des méta plans (ou stratégies). Le résultat sera déposé dans la structure afférente aux intentions. Si, parmi ces intentions, il s'en trouve une pour l'exécution immédiate d'une action atomique, l'agent l'exécute. Tous les éventuels événements externes générés pendant l'exécution d'un cycle seront ajoutés dans la queue

d'événements. Les événements internes sont ajoutés au moment de leur création. Ensuite, l'agent modifie les structures d'intentions et de buts en effaçant les buts et les intentions satisfaites, ainsi que les buts impossibles ou les intentions irréalisables.
Un module de planification analyse les informations contenues dans le modèle de l'environnement. Il définit une séquence d'action à exécuter pour atteindre les buts désirés par une inférence sur ses connaissances. Une fois la séquence connue (le plan) un module transforme ces plans de haut niveau en une série de primitives d'actions et de perceptions.
Du fait de ce raisonnement à haut niveau, les contraintes temporelles par rapport à la dynamique de l'environnement posent de sévèrement problèmes.
L'arbitrage entre précision[2] et temps de calcul est critique voire impossible à trouver du fait de la complexité des algorithmes de manipulation symbolique. Plusieurs modèles et méthodes de planifications associées ont été proposés pour répondre aux questions de la représentation de l'environnement et de la politique optimale tel que les modèles basés sur l'historique, les MDP (Markov Decision Process), et POMDPs [Kaebling & all, 98] (les processus de décision markovien partiellement observables). Mais ils souffrent de plusieurs inconvénients, dont les plus importants sont : la complexité NP-Hard, l'explosion combinatoire de la taille de l'espace des états, et la dépendance de l'agent vis-à-vis d'un oracle qui a une vision totale de l'environnement pour construire le modèle.

1.3.2 Agent réactif

Ceci a conduit au paradigme réactif pour lequel une activité intelligente - dans le sens où elle est pertinente dans l'environnement courant pour le but visé- peut émerger de la collaboration de multiples composants très simples. Il n'y a pas de modélisation interne du monde extérieur, source d'incohérences car souvent imparfaite : suivant Gibson [Gibson, 86], le monde est sa propre meilleure représentation. L'intelligence est dans l'œil de l'observateur, ce n'est pas une propriété isolée [Brooks 91]. Cependant cette émergence est difficile à obtenir : elle requière de nombreux essais/erreurs. Un agent réactif possède une hiérarchie de comportements regroupés par niveaux : le bas niveau émule les fonctions réflexes, tandis que le haut niveau modélise des fonctions complexes – en particulier il peut accéder aux sorties du bas niveau pour ses calculs. Chaque couche de l'agent est en contact avec ses capteurs et ses actionneurs et génère des réponses en fonction des stimuli reçus. Un agent est alors constitué d'un ensemble de comportements. Chaque comportement est une machine à états finis. Les agents purement réactifs ne tiennent nullement compte du passé : leurs actions sont justes basées sur la simple perception du présent qui entraîne le passage à l'acte. C'est la réponse du paradigme réactif à la dualité précision/temps de calcul. De plus, le haut niveau a la priorité pour subsumer le rôle des niveaux inférieurs lorsqu'il prend le contrôle et peut inhiber les entrées ou supprimer les sorties du bas niveau. Il n'est pas nécessaire de modifier le niveau inférieur lorsque l'on ajoute un comportement plus complexe (approche incrémentale).
Ceci conduit rarement à un comportement optimal du robot : ses perceptions limitées le guident à court terme et il ne reconnaît pas les situations d'échec, ce qui peut le mener à des impasses (absence d'apprentissage).
A noter que le comportement d'un agent réactif est déterministe, celui d'un agent cognitif ne l'est pas véritablement puisque son état mental est une boite noire (sauf peut être pour le concepteur). Par contre le système composé d'agents réactif n'est pas déterministe.

1.3.3 Agent hybride

Bien qu'apparemment diamétralement opposées, les deux approches précédantes peuvent être vues comme étant complémentaires.
Le modèle hybride fait cohabiter un niveau délibératif et un niveau réactif. Ils ne fonctionnent pas au même rythme : l'action instantanée pour le réactif, la planification sur la longueur pour le délibératif. Il y a donc au minimum deux couches dans une architecture d'agent hybride mais il est possible d'avoir une hiérarchie complexe de couches qui interagissent entre elles. Notamment les architectures de contrôle multi-agents comme Cypress [Wilkins & all, 95] et RAP [Firby, 89] (Reactive Action Package) qui distinguent des aptitudes de bas niveau et les raisonnements de haut niveau. Dans une telle architecture, un agent est composé de modules qui gèrent indépendamment la partie réflexe (réactive) et réfléchie (cognitive) du comportement de l'agent. Le problème central reste de trouver le mécanisme idéal de contrôle assurant un bon équilibre et une bonne coordination entre ces modules.

[2] L'ouvrage de WEISS [Weiss, 99] fait une distinction entre agents logiques qui sont peu adaptés à la robotique et les agents BDI qui sont utilisés avec succès en robotique.

Architecture du LAAS

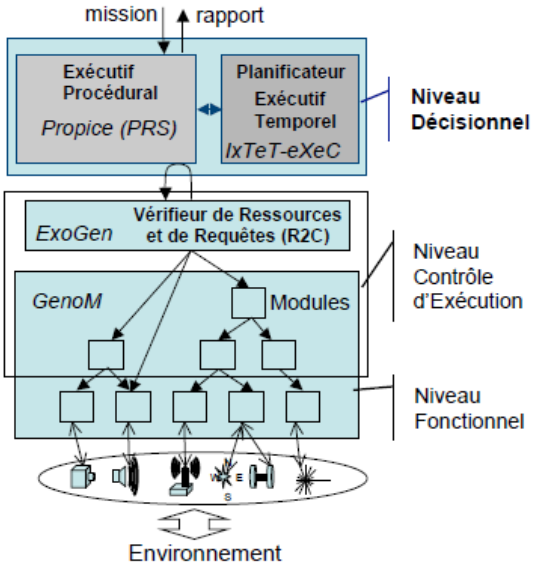


Figure 10 L'architecture LAAS (tiré de [Ingrand, 05])

L'architecture LAAS est composée de trois niveaux. Chacun d'entre eux a ses propres contraintes temporelles et représentations de l'état du système.
Le niveau décisionnel comprend un exécutif procédural (PRS/Propice [Ingrand & all, 96]) connecté au niveau inférieur à temps de réaction garanti. Il a aussi un planificateur/exécutif temporel IXTETEXEC (extension de IXTET [Ghallab & all, 94]) chargé de produire et exécuter des plans temporels.
Le niveau fonctionnel est l'interface entre les composants des couches supérieures et la partie physique du système.
Le niveau de contrôle des requêtes, situé entre les deux niveaux précédents, le R2C "Requests and Resources Checker" [Ingrand & all, 02] agit comme un filtre qui rejette éventuellement des requêtes en fonction de l'état et d'un modèle formel donné par l'opérateur spécifiant les états autorisés ou interdits.

Architetcture CLARAty

L'architecture CLARAty (Coupled Layer Architecture for Robotic Autonomy) [Volpe & all, 00] propose une architecture mono-robot complète. L'objectif principal derrière CLARAty est de développer une architecture pour des composantes robotiques génériques et réutilisables et qui peut être adaptée à diverses plateformes robotisées hétérogènes. En particulier, la couche décisionnelle est basée sur un principe important : la nécessité d'avoir un plan qui mêle une planification haut niveau pour le long terme, une planification fine à court terme et une zone en cours d'exécution qui ne peut pas être le sujet de la planification. Les séparations sont alors gérées en fonction de la situation : on réduira par exemple le court-terme dans une situation très changeante, par opposition à une situation très prévisible qui permettrait d'avoir un horizon plus long. L'implémentation actuelle utilise un couplage entre le planificateur CASPER [Chien & all, 00] pour le long terme et TDL [Simmons & all, 98] (Task Description Language) pour le court-terme, les deux outils étant unifiés par un superviseur chargé de régler les conflits entre ces deux composants et un protocole client serveur gère les interactions avec le niveau fonctionnel qui repose fortement sur une approche objets permettant une forte réutilisabilité du code et une extension aisée.

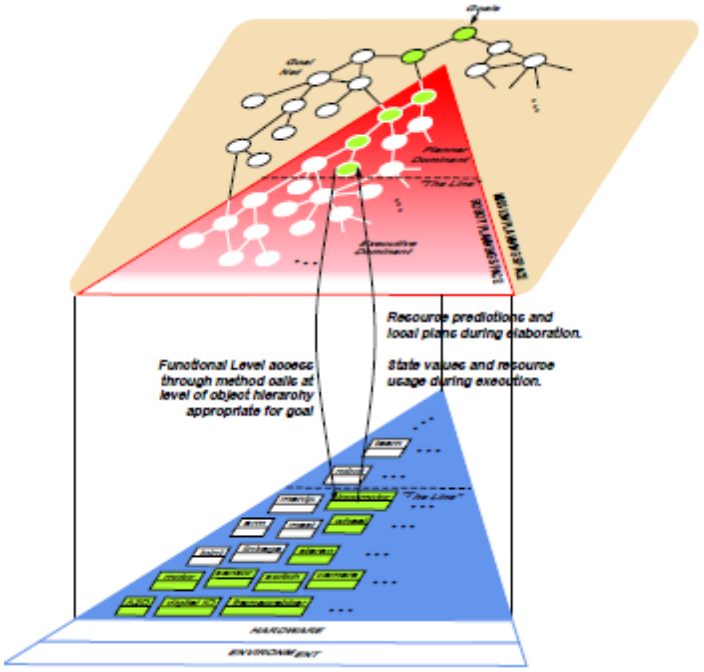


Figure 11 L'architecture CLARAty (tiré de [Ingrand, 05])

Architecture ARM

L'architecture ARM (Asynchronous Reflexive Model) [Malenfant & all, 03] prévoit deux niveaux dans l'architecture logicielle d'un composant robotique : le niveau de base qui correspond à la boucle réactive temps réél et le méta-niveau qui permet la délibération en adaptant le niveau de base par des notifications asynchrones. Ce modèle, défini dans le projet MAAM [Duhaut, 02] (Molecule = Atom | Atom + Molecule), introduit la possibilité d'adaptation du composant aux changements de son environnement et de ses buts. Le niveau de base, purement réactif, est celui qui réalise concrètement l'application (la mission) du robot. Le niveau délibératif peut être vu comme un niveau réflexif exerçant des calculs pour changer les réactions du niveau de base.

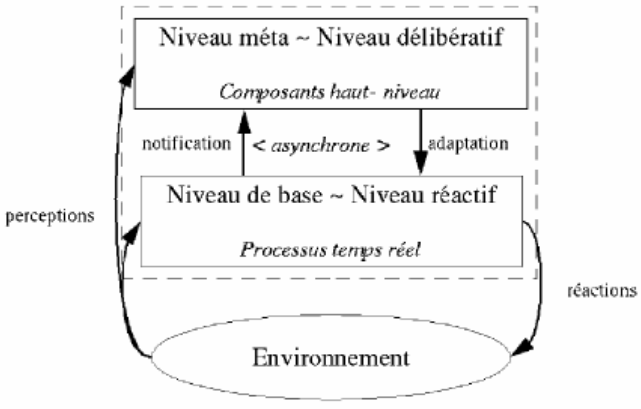


Figure 12 L'architecture ARM

Architecture ORCCAD

L'architecture ORCCAD [Borrelly & all, 98] (Open Robot Controller Computer Aided Design system) est un ensemble de concepts, de méthodes et d'outils destinés à spécifier, programmer, vérifier et implanter des applications robotiques. Les Tâches-Robot (actions de base) combinent une loi de commande et un comportement logique réactif et ainsi intègrent les aspects temps continu et discret. Ces actions sont ensuite composées de façon hiérarchique à l'aide du langage synchrone Esterel pour obtenir des actions de complexité croissante jusqu'à obtenir une application complète. La structuration rigoureuse du logiciel de contrôle/commande intègre des méthodes de vérification formelle au cycle de développement.

Architecture IDEA

L'architecture IDEA [Muscettola, 02] construit la décision comme une hiérarchie d'agents logiciels, chaque agent étant composé d'un plan, un planificateur et un exécutif, le modèle de plan étant unique à toute l'architecture. Ainsi l'agent robotique est constitué d'agents logiciels ! Il y a donc deux niveaux de description : le niveau méta-architecture qui correspond au robot et l'architecture des agents IDEA. L'interaction entre les agents se fait au niveau du plan, certains tokens (les éléments qui composent le plan) pouvant être pris en charge par des agents externes. Il n'y a donc pas de notion explicite de plan global. A noter que le langage de spécification des actions d'un agent est ESL [Gat, 97] (Executive Support Language) qui, s'appuyant sur les primitives du robot, spécifie comment obtenir le succès d'une tâche.

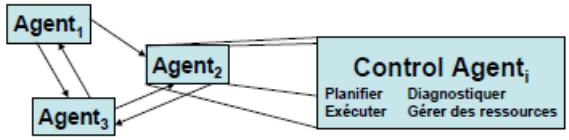


Figure 13 Collection d'Agents IDEA (tiré de [Ingrand, 05])

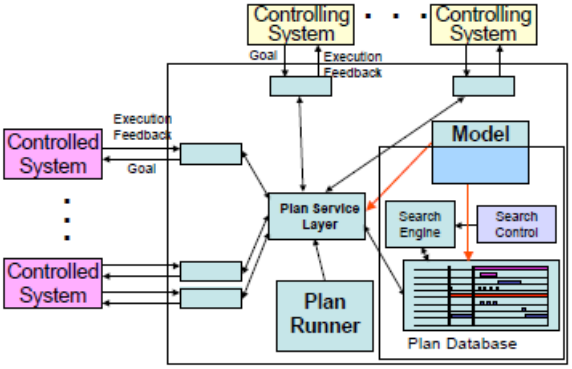


Figure 14 Structure d'un Agent IDEA

1.4 Systèmes Multi-Agents

Cette section a pour objectif de présenter les spécificités des systèmes multi-agents dans le cadre de la robotique. Elle permet de montrer que le langage MASL doit offrir un moyen de définir la coopération, communication et les interactions entre agents.

1.4.1 La coordination des agents

1.4.1.1 Le cas général

Les phénomènes d'interaction, de coordination, de coopération sont des concepts plus importants que l'agent lui-même. La présence d'un agent n'a de sens que lorsqu'il est plongé dans un environnement dans lequel évoluent d'autres agents, sinon, il est réduit à l'état d'un simple processus informatique communicant.
Les relations liant les agents aux autres agents sont donc primordiales par rapport à l'agent lui-même et à son comportement. Son comportement individuel étant plus la résultante de ses interactions en société.
Selon Weiss [Weiss, 99], la coordination est une propriété d'un système composé d'au moins deux agents, exécutant des actions dans un environnement partagé.
La coopération est une forme d'interaction. Elle consiste à établir qui fait quoi, avec quel moyens, de quelle manière et avec qui. Elle nécessite d'apporter des solutions aux différents sous-problèmes (la collaboration par répartition de tâches, la coordination d'actions et la résolution de conflits). La coopération se résume donc par la formule d'après Ferber [Ferber, 95] :
Coopération = collaboration + coordination d'actions + résolution de conflits
La collaboration est une forme d'interaction qui étudie la manière de répartir le travail entre plusieurs agents.
La coordination d'actions est l'ensemble des activités supplémentaires qu'il est nécessaire d'accomplir dans un environnement multi-agents et qu'un seul agent poursuivant les mêmes buts n'accomplirait pas. On peut distinguer la coordination par synchronisation, la coordination par planification, la coordination réactive et la coordination par réglementation. Elles s'intéressent à la manière dont les actions des agents sont organisées dans le temps et dans l'espace pour accomplir les buts.
Lorsqu'un conflit apparaît entre deux ou plusieurs agents, il y a deux méthodes pour le résoudre : l'arbitrage et la négociation.

L'arbitrage consiste à définir des règles de comportement qui agissent comme des contraintes sur l'ensemble des agents. Le résultat global a pour effet de limiter les conflits et donc empêcher la déstabilisation du système.
La résolution de conflit supposant une négociation pour lever ce conflit pour un agent cognitif. Pour un agent réactif, ce sont les règles aux niveaux des agents qui doivent l'éviter.

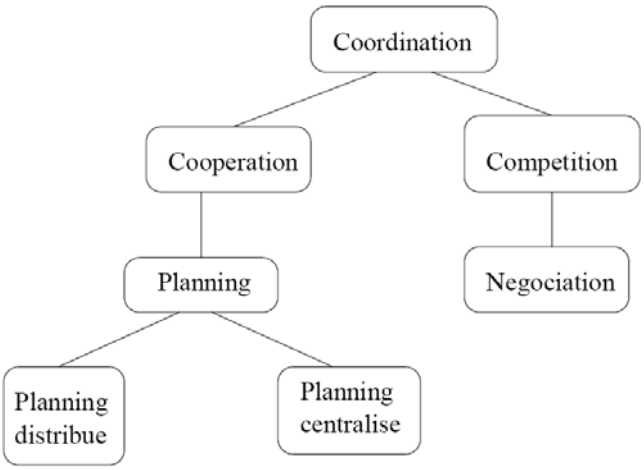


Figure 15 Schéma qui présente les relations entre différentes formes d'interaction selon [Weiss, 99].

1.4.1.2 Le cas de la robotique collective

Il existe de nombreux « survey » sur la robotique collective ([Arai & all, 02], [Dutta & all, 04]), certains débouchant sur des taxonomies suivies d'un classement des travaux évoqués dans la proposition des auteurs ([Cao & all, 97], [Iocchi & all, 01], [Farinelli & all, 04]). Concernant les taxonomies de la robotique en essaim, notons celles de [Dudek & all, 93] et [Bayindir & all, 07]. L'intérêt de ces taxonomies est alors de permettre la comparaison des travaux qui ne sont pas trop éloignés. Leur style est plutôt littéraire. Nous invitons le lecteur à les consulter. Enfin, signalons la compilation des modèles mathématiques du domaine dans [Gazi & all, 06]. L'article [Nembrini & all, 07] écrit en français explique les problématiques récentes de la robotique en essaim.
Par rapport aux agents en général, c'est-à-dire les agents logiciels, les agents robotiques utilisent des mécanismes et des interfaces physiques de communications dédiés, ce qui a pour effet que la communication n'est pas aussi évidente que pour les agents logiciels. De plus l'ordre de grandeur du nombre d'agents robotiques, même dans la robotique en essaim, est bien inférieur à celui que l'on peut rencontrer avec les agents logiciels.
La programmation d'un groupe voire d'une société d'agents [Klavins, 03] [Klavins, 04] [Mondada & all, 03] [Mackenzie & all, 97] nécessite des synchronisations pour réaliser la mission (foraging, déplacement en formations) ou pour se reconfigurer. Elle est rendue plus difficile si elle concerne des agents hétérogènes mêlant des matériels différents, des langages différents et des comportements différents.
Dans un système multi-robots, les problématiques mono-robots sont augmentées par les interactions entre robots. Potentiellement consommatrices de ressources, les interactions peuvent être mises à profit pour réaliser plus efficacement des tâches mono-robot (redondance, partages de parties de tâches), ou bien réaliser des tâches qui, par nature, ne pourraient être réalisées par un unique robot. Les mécanismes de coordination rendent cohérentes entre elles les actions des différents robots.
Mintzberg [Mintzberg, 79] identifie la coordination par ajustements mutuels, la coordination par supervision directe et la standardisation comme processus fondamentaux de coordination dans les organisations humaines. Les différents mécanismes de coordinations exploités dans les systèmes multi-robots sont tous des manifestations d'un ou de plusieurs de ces processus fondamentaux. Malone [Malone, 90] suggère qu'en utilisant ces processus de coordination fondamentaux "à la Mintzberg", il est possible de construire des systèmes de coordination sophistiqués. De nombreuses organisations mettent en oeuvre des processus de coordination mixtes basées sur l'ajustement mutuel, la supervision directe et la standardisation. Ces cadres organisationnels peuvent être des sources d'inspiration quand on doit établir une structure organisationnelle pour un SMA. Notons le caractère réactif de la standardisation. Par la suite nous reprenons pour l'enrichir, la présentation des mécanismes de coordination de [Gancet, 05].
Les approches émergentes définissent l'émergence de comportements cohérents de groupe, à partir de primitives comportementales individuelles simples. Les approches à base d'émergences ont pour principal avantage de permettre des exécutions flexibles et robustes : elles sont généralement employées dans un cadre réactif, pour des

missions ”unitaires”. Elles atteignent cependant leurs limites dans les applications complexes, où l’interaction nécessite une forme de concertation.

Respect de standard
Ce type d’approche repose sur la coordination par standardisation : les primitives comportementales sont en effet des formes de spécification de la conduite à tenir selon les situations rencontrées. Ces primitives peuvent être pré-définies dans le cadre d’un contexte particulier, ou bien apprises (les modalités d’apprentissage sont alors elles-mêmes des manifestations de la ”standardisation”).
Elle utilise une forme de communication dite indirecte, c'est-à-dire en relayant l’émetteur et le récepteur via l’environnement. On parle aussi de stigmergie. Il s’agit d’une forme particulièrement de coordination sans communication. C’est typiquement un comportement qu’adoptent, on ne sait pourquoi encore précisément, les bancs de sardines ou les vacanciers sur les plages au mois d’août. Ce comportement consiste simplement à se grouper en nombre. L’union faisant la force, ce comportement est utile pour se défendre contre la prédation. Le grégarisme est aussi utile pour la reproduction puisque facilite le choix d’un partenaire sexuel. Ainsi pour le déplacement collectif, peut on rencontrer des groupes structurés suivant la typologie proposée dans [Arnaud, 00] : en troupeau (herd), en volée ou en bande (flock), en banc (school), en formation, en front, en procession, en essaim (swarm), et en groupe non structuré (unstructured group). Par exemple, dans le cas des boids (cf. 1.4.1.3), les 3 règles de déplacements par rapport au voisin constituent le standard à respecter. Dans [Munoz, 03], la coopération est située : c’est le résultat des actions individuelles et collectives d’un système multi-agent, orientées vers la maintien de l’ensemble d’agents du système dans leur zone de viabilité. Ces actions sont entreprises grâce à un mécanisme d’adoption et procuration de buts entre les agents. Dans [Lucidarme & all, 02], la coopération repose sur la satisfaction personnelle d'un agent réactif qui représente son progrès dans la tâche demandée, et sur la satisfaction interactive de l'agent qui est positive si l'agent demande de l'aide, et négative s'il est gêné ou en conflit avec un autre agent. Les agents vont chercher à s'approcher ou s'éloigner de l'agent dont la valeur absolue de satisfaction interactive est la plus élevée. Il en résulte une combinaison des comportements centrés sur les désirs de l’agent avec des comportements coopératifs centrés sur les besoins des autres agents. Le standart de ce modèle peut se résumer ainsi : si les influences reçues (attraction / répulsion) sont plus intenses que la satisfaction personnelle, elles déclenchent des réactions altruistes.

Approche Equipes/Rôles
Dans l’approche équipes/rôles, des sous-groupes d’agents (équipes) sont formés pour répondre aux besoins de tâches jointes. Pour une tâche jointe donnée, un certain nombre de rôles sont définis dans le cadre d’une équipe : les membres de l’équipe doivent alors endosser ces rôles. Les rôles permettent d’expliciter les asymétries parmi les tâches ou fonctions devant être assumées pour réaliser la tâche jointe. Cette approche entre dans la coordination par standardisation, en considérant que les rôles sont prédéfinis. Cependant, on peut aussi considérer qu’elle fait appel à des mécanismes d’ajustement mutuels, pour le choix des rôles si par exemple on veut garantir que tous les rôles trouvent preneurs ou que l’on cherche à éviter la redondance.

Toujours dans cette approche, on peut ranger la très délibérative TOP [Tidhar, 93] (Team Oriented Programming) qui est une nuance de l’AOP (voir ci-dessous) dans laquelle la coordination est spécifiée du point de vue abstrait du groupe. Le comportement coordonné est programmé d’une perspective de haut niveau et le modèle d’exécution modifie les activités individuelles des agents concernés en conséquence. SimpleTeam [Yoshimura & all, 00] est une extension de la plateforme BDI JACK [Howden & all, 01] qui permet de spécifier le comportement coordonné de l’équipe sans préciser si ce comportement sera adopté par un seul agent ou par un ou plusieurs membres d’une équipe. Ce framework rajoute en particulier le concept de `team_plan` qui représente l’activité d’un sous groupe de l’équipe pour atteindre le but de l’équipe en ajoutant des primitives sur le contrôle de la concurrence et la gestion des exceptions. Toujours dans la Team Oriented Programming, Teamcore [Tambe & all, 99] et Machineta [Scerri & all, 04] font reposer la programmation de l’équipe hautement hétérogène sur des proxy agents semi-autonomes pour créer un niveau homogène de coordination au dessus des agents.

```
teamplan CaptureTargetPlan extends TeamPlan{
  #handles event InitiateCaptureEvent ice;
  #uses role NorthAgent north_agent;
  #uses role SouthAgent south_agent;
  #uses role EastAgent east_agent;
  #uses role WestAgent west_agent;
  body(){
    @parallel(){
      @team_achieve(north_agent,north_agent.pne.position_north_event());
      @team_achieve(south_agent,south_agent.pse.position_south_event());
      @team_achieve(east_agent,east_agent.pee.position_east_event());
      @team_achieve(west_agent,west_agent.pwe.position_west_event());
    };
  @team_achieve(north_agent,north_agent.rfce.ready_for_capturing());
```

```
}
```

Figure 16 Exemple de plan dans SimpleTeam dans [Yoshimura & all, 00]

D'autres approches délibératives sont aussi envisageables :

Contrat

Les approches « à base de marché » se fondent sur des systèmes de contractants contractés, généralement inspirés du CNP (Contract-Net Protocol) [Smith, 80] et s'inscrivent typiquement dans la coordination par ajustements mutuels. Le CNP est un protocole de haut niveau pour la communication entre les noeuds d'un résolveur de problèmes distribués. Il facilite le contrôle distribué de l'exécution de tâches coopératives avec une communication efficace entre les noeuds.

Le CNP s'intéresse à l'allocation de tâches dans un réseau : un manager veut déléguer une tâche et fait un appel d'offres pour connaître les différents noeuds du réseau prêts à l'effectuer pour son compte. Il collecte les offres des différents contractants et choisit celle qui le satisfait le mieux. La tâche est allouée au contractant associé à la meilleure offre. Le manager attend ensuite les résultats de l'exécution de cette tâche. Ce protocole est à la base quasiment de tous les travaux sur les protocoles de négociation.

Les approches « à base de marché » sont de plus en plus exploitées en robotique, principalement pour répondre à la problématique de l'allocation de tâches dans un système multi robots distribué. Elles confèrent beaucoup de souplesse à l'allocation de tâche, bien que sous optimales. Elles supposent cependant que la décomposition d'une tâche de haut niveau en tâches distribuables est déjà réalisée, et que la décomposition de tâches en sous tâches est indépendante de l'affectation. Elles supposent également que les tâches à distribuer peuvent systématiquement être évaluées par les contracteurs potentiels.

Si cette approche a démontré qu'il était possible de réaliser une allocation de tâche sans connaissance des plans des autres membres de l'équipe, des mécanismes plus avancés de coopération comme l'opportunisme et les mécanismes de planification multi-robot ont besoin de partager des informations de manière plus extensive. Son usage ne se conçoit que dans une situation où les objectifs sont partagés par différents agents. De plus, Krishna et Ramesh [Krishna et al. 98] soulignent que le protocole CNP n'est utilisable que dans le cas où les agents ont un but commun et ne l'est plus dès lors que les agents ont des objectifs conflictuels.

A noter que l'approche contractuelle [Dias & all, 06] est aussi utilisée pour l'allocation de tâche pour la robotique en essaim.

Supervision

Les approches orientées « planification centralisée » qui reposent sur la supervision, permettent la synthèse d'un plan global pour une mission donnée comme l'union d'un ensemble de plans des robots du système. Les individus du système sont modélisés centralement, et la centralisation des connaissances permet de construire le plan global « le mieux informé » (en principe), comparé à la plupart des alternatives distribuées. Elles permettent aussi de faire face à des problèmes complexes, bien qu'elles se heurtent à des problèmes « d'échelle » (scalability) : la centralisation des informations est en effet susceptible d'être à l'origine d'un goulot d'étranglement du transit des données.

Dans le cas de la planification distribuée incrémentale, suite à l'acquisition de plan de chaque agent du système, la fusion de plans intervient à posteriori, comme validation en un plan global par un mécanisme de coordination où, peu avant l'exécution effective d'une partie du plan, chaque robot réclame l'autorisation de manipuler le plan global. Une fois l'autorisation obtenue, il insère dans le plan global ses propres tâches, ainsi que des contraintes d'ordre destinées à prévenir les conflits possibles entre ses tâches et le plan global déjà existant. Les autres robots devront ensuite tenir compte de ces contraintes, avant d'insérer à leur tour leur propres tâches et contraintes dans le plan global, incrémentalement. Cette approche est particulièrement utile pour prévenir les conflits de ressources entre des robots. [Joyeux, 07] propose une architecture de supervision permettant l'intégration de producteurs de plans hétérogènes - y compris des producteurs de plan multi-robot - avec des mécanismes de supervision. L'approche par fusion de plans correspond aux ajustements mutuels.

Dans l'approche totalement distribuée, asynchrone, la coordination de plans et décomposition des tâches vont de concert par le biais d'un flux de communications élevé, afin de transmettre des requêtes informatives aux autres individus et de traiter les requêtes reçues. Le protocole peut inclure des notions d'intention et d'engagement, grâce auxquelles les robots peuvent prendre en considération dans leur propre plan les effets des activités des autres robots. Cette récente approche est ambitieuse, et son applicabilité sur robots réels n'est pas encore démontrée. Elle se situe dans le cadre de coordination par ajustements mutuels. Cette approche est en particulier présentée dans [Brenner, 03].

1.4.1.3 Quelques exemples de la robotique collective

Parmi les applications classiques de la robotique collective on peut citer de façon non exhaustive : la robotique en essaim, la robotique modulaire et la robotique collaborative.
La robotique en essaim repose sur une coordination auto organisée de systèmes multi-robots, et tout particulièrement à ceux qui consistent en un grand nombre d'unités dont la complexité est minimisée. Cette simplicité visée de l'élément robotique individuel permet d'envisager une miniaturisation, des réduction de coûts, une dégradation progressive plutôt que soudaine de la performance collective optimale face à un changement dynamique du nombre de robots, ainsi qu'une robustesse améliorée au niveau du système mais aussi des robots individuels.
La robotique modulaire minimise elle aussi les coûts des unités. Les reconfigurations dynamiques des groupes aboutit à des formes différentes du robot agrégat. La frontière entre les deux premières est de plus en plus ténue.
La robotique collaborative est centrée sur l'équipe. Chaque membre concoure à l'objectif de son équipe comme pour l'application des robots footballeurs.

Les problèmes de déplacement en groupes

Les bancs de poissons ou d'oiseaux Boids Craig Reynolds, 1986 [Reynols, 86]
Les 3 règles utilisées sont :
• Eviter les collisions avec ses voisins en s'écartant quand on est trop proche;
• Aller à la même vitesse (proche de la vitesse moyenne) et dans la même direction (proche de la vitesse moyenne);
• Rester à proximité de ses voisins.
Si tous les oiseaux appliquent la même politique, le groupe d'oiseaux présente alors un comportement émergent de vol, sous forme d'un corps pseudo rigide de densité à peu près constante, qui ne se disperse pas avec le temps.

Les robots de type chaîne

Les agents forment une chaîne, parfois reconfigurable, où la principale problématique est la coordination des efforts sur une structure fixe Conro (EU) [Rubenstein & all, 04], M-Tran (Japon) [Matura & all, 02], PolyPod (Xerox) [Polypod, 97], PolyBot (Xerox) [PolyBot, 98]

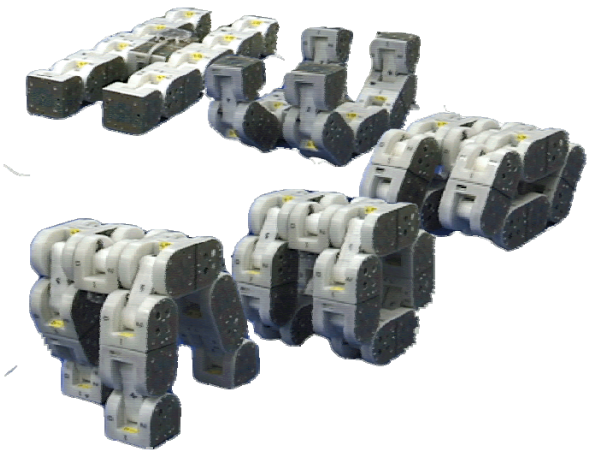


Figure 17 Le système reconfigurable M-TRAN

Robots Treillis

Les agents sont des modules qui se déplacent les uns par rapport aux autres. La principale problématique est la reconfiguration. Atron (Danemark) [Jorgensen & all, 04], Molecule (EU) [Rus & all, 02], TeleCube (Xerox)

[Kubika & all, 01], Proteo (Xerox) [Lamping & all, 97]. Ils ont les mêmes caractéristiques qu'un robot type chaîne :
• possibilité de s'attacher et se détacher d'un autre module ;
• un module reste toujours accroché à l'entité principale ;
• les modules se déplacent selon un treillis (damier) en 2 ou 3D donc les déplacements sont plus contraints.

Figure 18 Le système reconfigurable ATRON

Composants autonomes
Projet Swarm-Bots (à roues plutôt chenille) [Dorigo & all, 04]

Un système robotique d'essaim est normalement caractérisé par beaucoup d'individus, chacun avec une connaissance partielle/limitée du modèle global dont il constitue un élément. Dans de tels systèmes les décisions collectives posent de grandes problématiques.
L'objectif scientifique principal du projet de swarm-bot est d'étudier une approche de la conception et de l'exécution des objets partant de l'auto-organisation et de l'auto-reconfiguration. Cette approche trouve ses racines théoriques dans des études récentes sur l'intelligence d'essaim montrées par les insectes sociaux et d'autres sociétés animales [Chauvin, 82] [Lorenz, 84].

Un swarm-bot comporte des robots mobiles autonomes appelés les s-bots. Les s-bots peuvent agir de façon individuelle ou sous la forme d'un collectif auto reconfigurable dans un swarm-bot en utilisant leurs pinces.

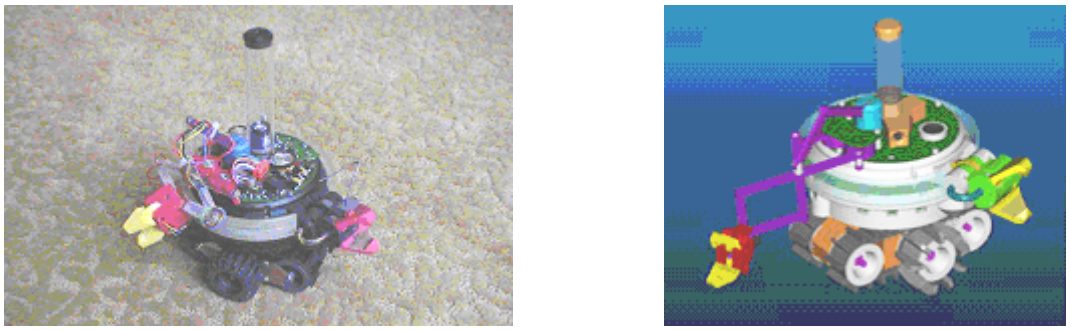


Figure 19 Le S-bot, réalisation matérielle et son modèle CAO

Les swarm-bot sont des robots ayant comme particularité l'auto-assemblage de ses modules. C'est un concept partant de la robotique distribuée et qui se trouve à l'intersection entre la robotique collective et reconfigurable.

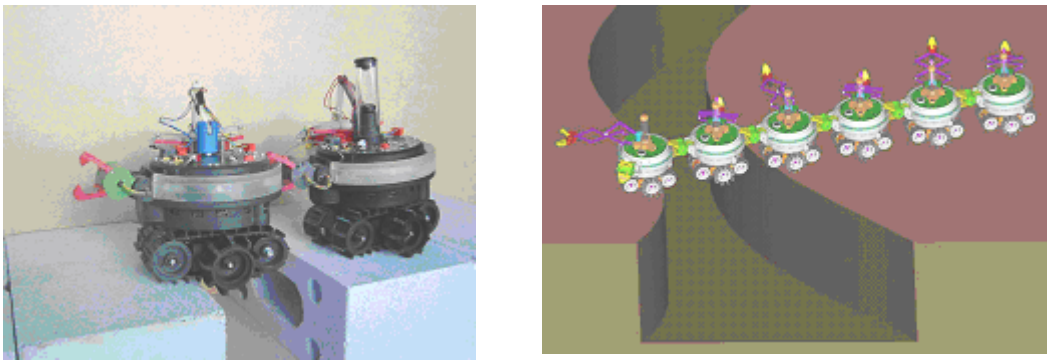


Figure 20 Le Swarm-bot franchissant un fossé

La séquence de la Figure 20 montre qu'un s-bots seul ne peut pas franchir un obstacle, mais après une reconfiguration en Swarm-bot de deux s-bots en ligne l'obstacle est franchissable.

Le travail est sous l'effet coordonné de quatre activités principales : la conception et l'exécution du matériel, la conception et l'exécution du simulateur, la conception et l'exécution du système de commande du Swarm-Bot, l'intégration, l'essai et l'évaluation des résultats de ces activités. Basés sur de telles caractéristiques des prototypes seront développés, évalués et documentés pour chacun des trois composants: s-bots (matériel), simulation (logiciel), et mécanismes de commande basée sur l'intelligence collective (logiciel). Les mécanismes de commande sont testés sur le simulateur et les réalisations de matériel.

Le projet MAAM qui est à l'origine du travail sur MASL

Le projet MAAM [Duhaut, 02] (Molecule = Atom | Atom + Molecule) a pour objectif de définir, spécifier, concevoir et réaliser un ensemble d'atomes robotiques capables de s'assembler en une molécule qui pourra, par reconfiguration successive, réaliser une tâche donnée. Dans le cas du projet MAAM, un atome sera une structure mécanique à six pattes, chacune d'elles pouvant se solidariser deux à deux avec d'autres atomes. Les six pattes sont réparties autour d'un noyau suivant les trois directions de l'espace comme sur la Figure 21. Les pattes sont munies de récepteurs et d'émetteurs infrarouges, ce qui permet à un atome de communiquer avec ses voisins (Figure 24). La partie centrale est munie d'une interface de communication Bluetooth qui dote le composant d'une communication radio courte distance.

Un des buts de la robotique reconfigurable du projet MAAM est d'arriver à obtenir des robots composés de plusieurs unités qui par le biais de reconfigurations dynamiques peuvent évoluer à l'instar des cellules vivantes s'auto-organisant pour former des organes, des tissus, etc. De même, la reconfiguration des modules du robot permettrait l'adaptation à l'environnement et de varier sa forme selon les tâches qu'on lui attribue.

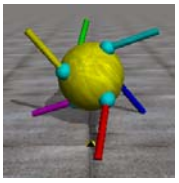


Figure 21 MAAM comme composant autonome

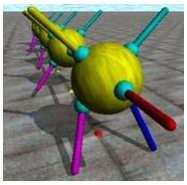


Figure 22 MAAM comme chaîne

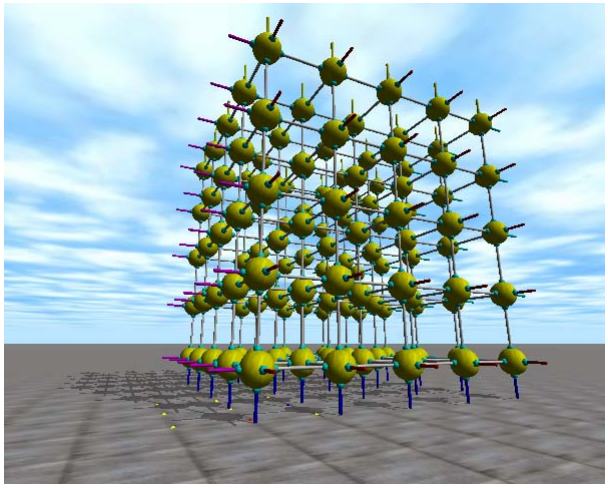


Figure 23 MAAM comme treillis

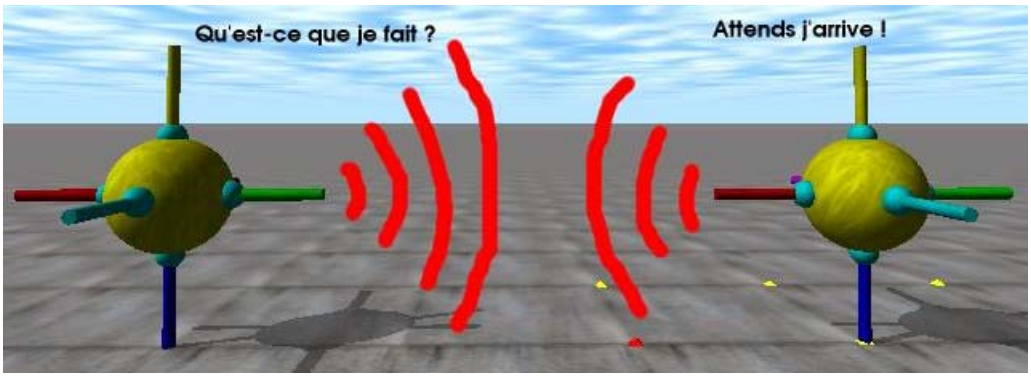


Figure 24 La communication et la coopération entre deux atomes MAAM

Pour avancer, l'atome seul peut rouler. Au début, Il est en équilibre sur trois pattes posées au sol. Il faut calculer alors la patte la plus éloignée, sur laquelle on applique ensuite une force qui tire la patte vers l'atome. Les deux autres pattes sont poussées pour être le plus près possible du sol. Si les pattes sont suffisamment longues, l'atome effectue alors une rotation comme le montre la Figure 25.

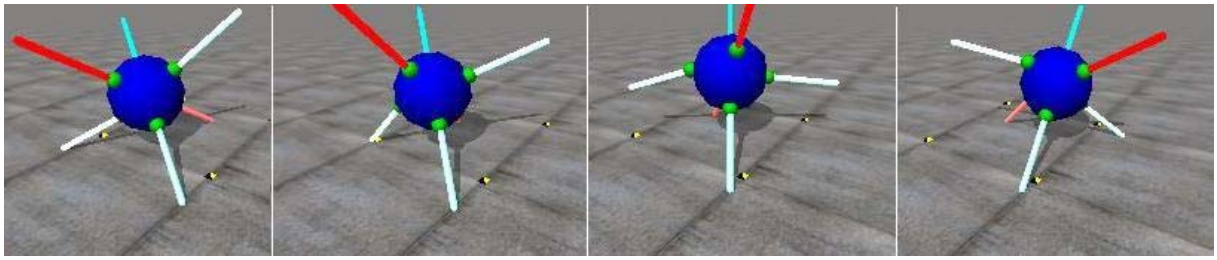


Figure 25 L'atome MAAM rouleur

Le déplacement collectif du robot reconfigurable se fait par des reconfigurations successives de ses modules. La reconstruction physique d'une molécule par re-connexion entre deux atomes robots distant est une opération longue car elle exige de la communication, de la localisation et de la coopération. Par rapport au projet

swarmbots, l'algorithme de marche d'un atome seul (sur trois pattes) n'est pas aussi trivial que pour des robots à roues ou à chenilles. De plus la connexion entre deux atomes proche est plus contraignante qu'entre deux s-bots.

Le projet swarmanoids

Les "swarmanoids" constituent eux aussi un groupe de machines en relation les unes avec les autres, mais cette fois-ci, une différence majeure par rapport au projet swarmbots va être introduite : les membres de la tribu seront désormais spécialisés. Il y aura des robots qui se consacrent aux déplacements, d'autres possèdent des mains leur permettant de grimper ou de saisir des objets, une troisième espèce, enfin, disposent d'yeux permettant de superviser le déroulement des opérations. Tous les robots ont la possibilité de communiquer entre eux par communication sans fils. Les Foot-bots sont de fait des s-bots améliorés. La même démarche que celle du projet swarmbots (son prédécesseur) est adoptée : le couplage simulation/vérification matérielle.

Figure 26 Un essaim hétérogène du projet swarmanoids

Figure 27 Foot-bots du projet swarmanoids

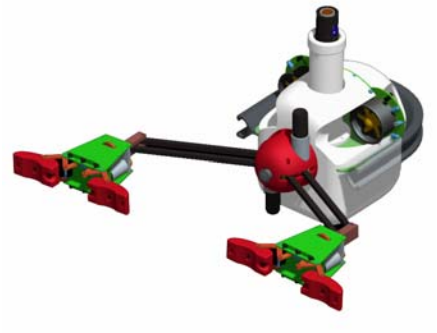


Figure 28 Hand-bots du projet swarmanoids

Figure 29 Eyes-bots du projet swarmanoids

Par rapport au projet MAAM, le projet swarmanoids a en commun des communications entre modules. Seules les foot-bots peuvent s'agréger ce qui fait que les composants autonomes à capacité d'agrégation sont eux homogènes comme pour le projet MAAM.

Les robots footballeurs

La Robocup [Kitano & all, 97] est une compétition internationale de robots footballeurs, qui a lieu chaque année, et dont l'ultime but est de former une équipe de robots humanoïdes autonomes capable de gagner contre l'équipe humaine championne du monde de football en 2050. Plusieurs ligues coexistent : la Middle-Sized Soccer Robot League, la Small-Sized Soccer Robot League (Figure 32) et la Sony Four Legged Robot League (Figure 31) qui mettent en jeu des robots physiques, et la Simulation Soccer League (Figure 34) qui relève uniquement de la simulation, les robots étant simulés sur un terrain virtuel. A partir de l'édition 2008, Nao est la plate-forme officielle de la ligue standard de la Robocup, en remplacement du robot de Sony.

Le but ultime de cette compétition est donc : "*By 2050, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official FIFA rules, against the winner of the most recent World Cup of Human Soccer.*"

Les équipes de la RoboCup [Iocchi & all, 01] ont fait le choix des approches délibératives ou réactives pour leur système multi robots (MRS) principalement en fonction des capacités possibles au niveau des compétitions. La vision globale autorisée dans la ligue Small-Size a conduit à des approches centralisées et délibératives. Dans la ligue Middle-Size où les robots sont pleinement autonomes, les approches réactives et délibératives sont présentes même si ce sont les dernières qui semblent préférées. Pour la ligue Sony Legged, on ne rencontre que des approches réactives du fait de la difficulté de communication entre robots.
La problématique de la RoboCup a été étudiée particulièrement dans l'introduction.
Dans [Duhaut & all, 08], MASL est présenté dans le contexte de la RoboCup. Il en est de même dans le chapitre [Dubois & all, 08].

Figure 30 Le robot Nao pour l'édition de la ligue standard de la RoboCup 2008

Figure 31 Ligue de robots SONY AIBO

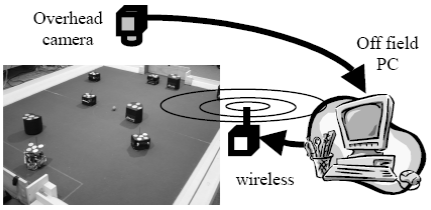


Figure 32 Principe de la ligue SMALL-Size

Figure 33 Ligue Middle-Size

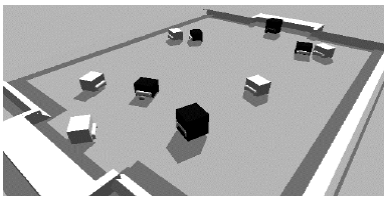


Figure 34 La ligue de simulation (ici de la Small-Size)

1.4.1.4 Adaptation des architectures dans le contexte multi robots

Les architectures logicielles présentées dans le paragraphe 1.3 concernent un seul agent. A noter que l'architecture LAAS prévoit la communication avec d'autres robots. Des architectures spécifiques pour la coopération entre robots ont aussi été proposées. Pour une étude plus poussée, voir [Beaudry, 05].

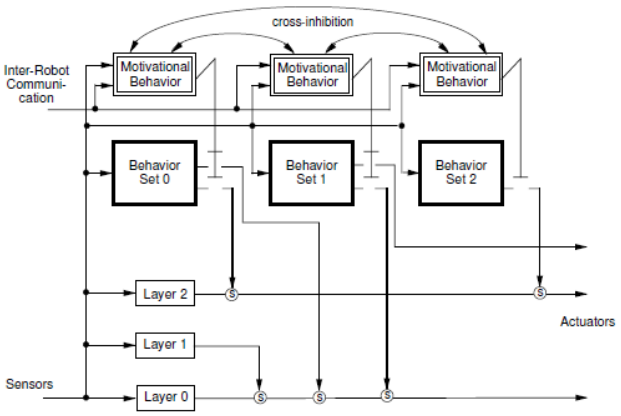


Figure 35 Architecture ALLIANCE

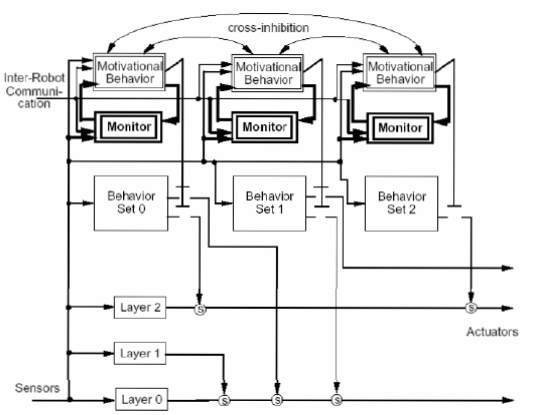


Figure 36 Architecture L-ALLIANCE

L'architecture ALLIANCE [Parker, 98] a été développée dans le but de permettre la coopération au sein de systèmes multi-robots hétérogènes. Elle est une architecture distribuée qui impose certaines règles d'implantation de façon à assurer une certaine tolérance aux fautes. La Figure 35 résume schématiquement le fonctionnement de cette architecture. L'architecture ALLIANCE implémente 3 niveaux selon un principe :

- au niveau 1, des comportements réactifs répondent au principe de la Subsumption Architecture ;
- le principe de Behavior Set (niveau 2) contrôle l'activation ou l'hibernation de groupes de comportements situés au premier niveau àfin que l'architecture se reconfigure dynamiquement ;
- le principe de Motivational Behavior (nivau 3), active / désactive le deuxième niveau tout en considèrant en plus des informations sensorielles, des informations de motivation interne permettant entre autre l'inhibition latérale des motivations (cross-inhibition), et finalement des informations provenant de la communication inter-robots pour une coopération multi-robots.

Une extension de l'architecture ALLIANCE, nommée L-ALLIANCE [Parker, 00], a été développée de façon à permettre l'implantation de mécanismes d'apprentissage par renforcement (« reinforcement learning ») au sein de l'architecture. En observant la Figure 36 on remarque la distinction avec l'architecture originale, qui correspond à l'ajout du principe de Monitor. Brièvement, le Monitor est en mesure d'influencer les comportements de la troisième couche, celle des Motivational Behavior en fonction de seuils de réaction.

Layered Architecture [Simmons & all, 02] permet à plusieurs robots de coordonner explicitement leurs actions selon plusieurs niveaux d'abstraction. Chacune des trois couches (Figure 37) peut discuter avec son homologue dans un robot différent. L'interaction est définie directement au niveau comportemental, au niveau d'exécution (supervision) ou au niveau planification. A chaque niveau, les robots utilisent des comportements coordonnés, des exécutions coordonnées de tâches et une planification coordonnées.

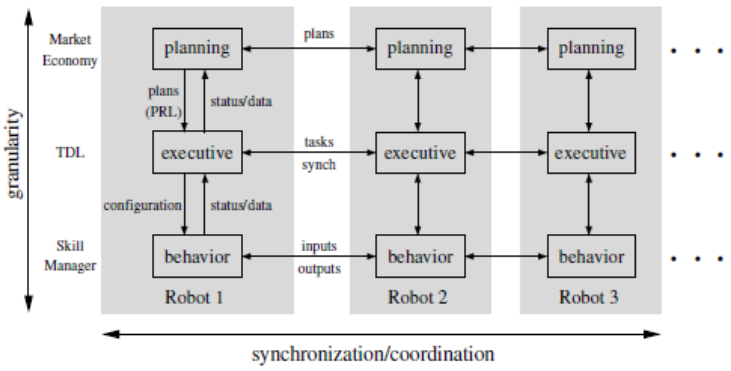


Figure 37 L'architecture à couches

CAMPOUT [Huntsberger & all, 03] (Control Architecture for Multirobot Planetary OUTposts) est une architecture qui a été développée par la NASA au Jet Propulsion Laboratory du California Institute of Technology. L'architecture proposée par CAMPOUT est une architecture hybride à trois couches distinctes où perception et cognition (prise de décision) se font de façon entièrement, et strictement, distribuée au sein du système multi-robots.

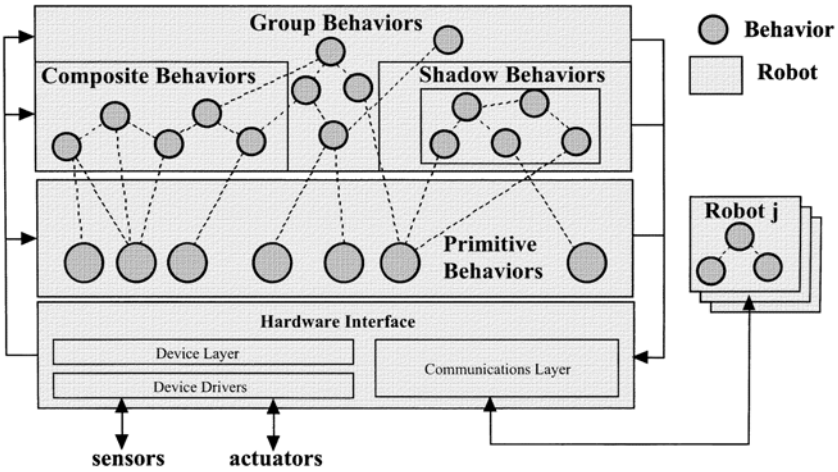


Figure 38 L'architecture CAMPOUT

La couche supérieure permet la planification, l'allocation et le monitorage hiérarchiques de tâches : elle gère l'utilisation des divers comportements. La seconde couche permet la composition de différentes classes de comportements. La dernière couche de cette architecture permet de faire le lien entre la couche des comportements et le matériel du système robotisé (capteurs et actuateurs). La Figure 38 schématise le fonctionnement de l'architecture de CAMPOUT. On peut y retrouver les différentes classes de comportements ainsi que les différents outils disponibles pour chacune des tâches.
L'architecture intègre une infrastructure de communication (basée sur des sockets TCP/IP), propose trois classes de comportements :

- les comportements de communication (Communication Behaviors) ;
- les comportements images (Shadow Behaviors) pour la représentation des comportements des coéquipiers
- les comportements de coopération/coordination (Cooperation/Coordination Behaviors).

Un robot du système peut composer des comportements primitifs (Primitive Behaviors) en des comportements composés (Composite Behaviors).

1.4.2 Plateformes SMA

Il existe de nombreuses plateformes de systèmes multi-agents mais elles représente ce que nous ne souhaitons pas faire dans la mesure où elle implémentent souvent une implémentation particulière des agents ou un framework d'objets qui ne font que facilité la programmation des agents. MASL se différencie de ces plateforme par son indépendance par rapport aux types d'agents.

Plateformes réactive et génériques

La mise en œuvre d'expérimentations en conditions réelles peut s'avérer longue et coûteuse. Les plateformes multi-agents pour la modélisation de la dynamique des systèmes complexes permettent, en s'affranchissant des contraintes techniques liées au monde réel, de disposer d'un cadre stable et convivial pour procéder à des simulations contrôlées et reproductibles. Des plateformes génériques (SWARM [Minar & all, 96], REPAST [Collier, 2002]) ne font pas d'hypothèses sur les modèles de systèmes complexes et fournissent l'accès aux agents via des langages de programmation généralistes comme C++ ou java. Ce sont des frameworks purement réactifs. Ils mettent aussi à disposition des bibliothèques d'algorithmes génétiques pour obtenir l'émergence requise du système. L'approche fonctionnelle permet aussi de créer des systèmes réactifs en évaluant, d'une manière déclarative, les entrées pour créer des sorties : on parle alors du paradigme FRP [Wan & all, 00] (Functional Reactive Programming). Des événements continus et discrets peuvent déclencher des actions : le modèle de temps est continu. FRAN [Elliott & all, 97] (Functional Reactive Animation) et FRP sont des langages flots de données avec des aspects dynamiques sous forme de bibliothèques Haskell [Wan & all, 00].

Les constructions `behaviours` et `events` permettent de modéliser des systèmes hybrides mêlant temps continu et temps discret. Les comportements sont des séquences continues qui varient en fonction du temps tandis que les évènements peuvent être des occurrences continues ou discrètes.

En opposition, des plateformes spécifiques (StarLogo [Resnick, 94 ;Tissue & all, 04], MadKit [Gutknecht & all, 01], Breve [Klein, 02], MissionLab [MacKenzie & all, 97], [Endo & all, 04]) exploitent un ensemble de concepts.

Plateformes spécifiques à temps constant

StarLogo, MadKit proposent un environnement de simulation à pas constant.

Plateformes spécifiques à temps continu

En permettant plusieurs collisions entre deux itérations de la simulation physique, Breve utilise un temps continu qui améliore la précision de la dynamique mais la limite à un petit nombre d'agents. Bien qu'un agent puisse être plongé dans un environnement possédant une dynamique modélisée de manière discrète ou continue, son processus décisionnel fait évoluer ses variables d'état de manière discrète.

Plateformes spécifiques militaires

Le modèle HAC (Hierarchical Agent Control) [Atkin & all, 99] pour le domaine militaire est centré sur une hiérarchie d'actions, de capteurs et de contextes (voir les Figure 65, Figure 66, Figure 67 qui concernent les explications sur TAPIR, successeur de HAC). Les actions abstraites sont exprimées à l'aide d'actions plus simples jusqu'à ce que cela se traduise par des primitives sur des actionneurs. Il existe de même des capteurs abstraits qui seront utilisées par les actions abstraites. Les contextes permettent d'interpréter les informations en fonction des buts à atteindre. Les actions sont des processus continus qui sont guidés par les buts et qui rendent compte à leur unique parent : les buts et les contextes descendent et les informations remontent le long de la hiérarchie. Ces hiérarchies permettent d'avoir des actions délibératives et réactives dans un même formalisme grâce à un gestionnaire de plans d'exécution qui est intégré dans la boucle réactive.

MissionLab est une plate-forme dédiée robotique mobile collective qui permet à la fois de réaliser des simulations et de contrôler des robots réels. Les comportements spécifiés de manière graphique peuvent être exécutés en simulation ou sur des robots réels.

1.5 ACL et FIPA

Il est difficile de parler de système multi-agents sans présenter ces travaux qui ont comme objectif de standardiser l'interopérabilité des agents. MASL n'a pas pour l'instant d'objectif de compatibilité avec ACL & FIPA car c'est un problème largement dépendant de l'implémentation et MASL se situe actuellement plus comme un langage de description de mission que comme un langage opérationnel avec un compilateur.

L'association de standardisation FIPA [Zhang & all, 98] (Foundation for Intelligent Physical Agents) fournit des standards pour faciliter l'interopérabilité entre les systèmes à bases d'agents. L'adjectif physique fait référence aux agents robotiques mais aussi à l'interaction homme-machine. Les concepts internes à l'agent ne sont pas fixés dans la norme.

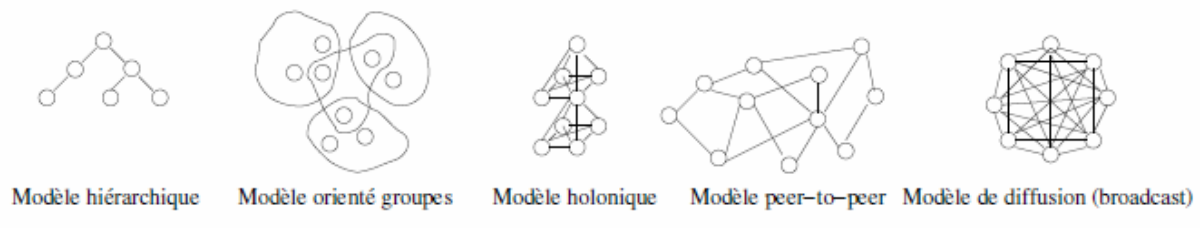


Figure 39 Les différentes topologies de communication

L'organisation des agents peut être caractérisée au niveau des communications par une structure topologique, c'est-à-dire la nature des liaisons entre les entités. La Figure 39 illustre cet aspect. Parmi les modèles les plus répandus, il y a le modèle hiérarchique (utilisé dans MRL, *Cf.* 1.6.3), le modèle orienté groupes (élément fondamental d'Aalaadin), le modèle holonique base de l'architecture InteRRaPR [Jung, 99] (une variante du modèle hiérarchique, dans lequel les fils d'un nœud sont totalement connectés[3]), le modèle peer-to-peer (comme dans la plateforme JADE) et le modèle de diffusion (broadcast).

Les agents coopèrent dans un ACL (Agent Communication Language) comme KQML [Finin & all, 97] ou FIPA-ACL [Labrou & all, 99]. Ici, les agents communiquent à un niveau élevé sur le monde ou sur les connaissances des agents, ce qui distingue les ACL de simples interactions entre objets : les messages sont basés sur la théorie des actes de parole, qui s'intéresse aux actions de communication comme informer, demander, offrir, accepter, rejeter et d'autres. Les agents qui sont autonomes, ont la liberté de décider s'ils vont ou non exécuter l'action spécifiée dans le message, par contraste à un objet passif.

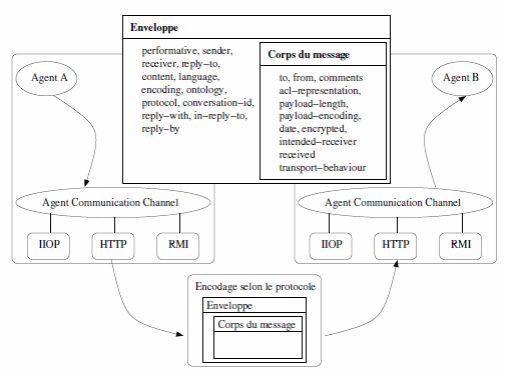


Figure 40 Envoi de message entre deux agents FIPA

La FIPA propose une bibliothèque de protocoles, destinés à encadrer les interactions dans certaines situations précises. Ces protocoles spécifient des actions autorisées suivant l'état de la conversation, et les réactions possibles à ces actions. Ainsi, lors d'une interaction protocolaire, chaque participant se conforme à sa partie du protocole, permettant une convergence rapide vers la résolution du problème auquel le protocole est dédié.

Plateformes basées sur FIPA

Les plateformes SAGE [Ghafoor & all, 04] (Scalable fAult tolerant agent Grooming Environment), JADE [Bellifemine & all, 03], JACK [Howden & all, 01] et sont conforme aux standard FIPA.
SAGE implémentent le modèle de référence FIPA en offrant l'infrastructure de communication obligatoire pour la composition, l'envoi et la réception des messages de l'ACL et des services (nommage, abonnement, facilitation, gestion des agents, outils de développement et de débogage.
Par défaut, les agents JADE ne sont que réactifs. L'extension JADEX [Pokahr & all, 05] représente les états mentaux de l'agent selon le modèle BDI. Les agents sont hybrides.
JACK est une autre plateforme FIPA compatible avec architecture BDI. La plateforme de simulation interactive oRis [Harrouet & all, 02], suite à certains développements pour des expérimentations au sein du laboratoire d'informatique industrielle de l'Ecole Nationale d'Ingénieurs de Brest, propose des paquetages de classes permettant la coordination d'action selon le Contract Net Protocol, la distribution d'agents, la communication entre agents en utilisant KQML. Si elle n'est pas encore compatible FIPA, elle est proche de l'être. Ainsi, la norme FIPA prend acte de l'absence de consensus au niveau des concepts internes des agents. De plus, il n'y a pas de méthodologie pour prouver toute l'architecture de système depuis sa spécification jusqu'à son implémentation.

[3] Holonique : derive du grec *holos* qui veut dire Tout et du suffixe *on* qui signifie partie.

1.6 Langages pour le contrôle de robots

Dans cette section nous introduisons un certain nombre de langages pour la programmation de robots. Nous pourrons ainsi vérifier qu'aucun d'eux n'offre simultanément les cinq propriétés autour desquelles est construit le langage MASL :

- Permettre de prendre en compte l'hétérogénéité des agents,
- Posséder des types d'exécution pour rythmer l'activité d'un groupe au sein d'une équipe,
- Proposer différents modes de communication entre robots,
- Introduire des « entrées dans le code » qui rendent simple les changements d'équipes,
- Offrir un dispositif dit de « perméabilité » qui permet la mise en œuvre de différents modes d'exécution.

Les langages comportementaux utilisent des méthodes de type réactif pour la programmation de robots. La syntaxe est proche de LISP, et les comportements sont représentés par un ensemble de règles qui sont compilés dans des machines à états finis représentant des fonctions dont les entrées/sorties peuvent être inhibées par les autres fonctions. Colbert [Konolige, 97] est un langage dans le style du langage C pour le contrôle de bas niveau dans l'architecture Sapphira dont la sémantique est elle aussi fondée sur des automates à états finis. De tels langages de bas niveaux ne proposent pas les abstractions que nous recherchons. Aussi exposerons nous de manière plus détaillés des langages intermédiaires dans lesquels on va spécifier la façon d'exécuter le plan. Notamment, PRS [Ingrand 96], ESL [Gat 97] et TDL [Simmons 98] sont des exécutifs robustes appartenant à cette catégorie. Il est aussi bon d'évoquer avant leurs ancêtres RAP [Firby, 89] (Reactive Action Packages) et RPL [McDermott, 93] (Reactive Plan Language). A noter que la question de savoir si ces langages intermédiaires sont multi-robots est secondaire. Il suffit de prévoir pour l'exécutif robuste d'un robot la possibilité de communiquer avec celui d'un autre robot.
D'autres langages considèrent la robotique comme un domaine hybride.
Enfin, nous étudierons les langages orientés agents de contrôle multi robots.

1.6.1 Langages intermédiaires de contrôle de robots, langages de supervision, de planification réactive.

Les langages sont appelés intermédiaires car ils se trouvent au niveau de la couche exécutive d'une architecture logicielle hybride à trois niveaux (entre la couche fonctionnelle et la couche décisionnelle). Ils peuvent aussi appartenir à des systèmes de planification réactive où l'agent réflexe, éventuellement doté d'un état interne, est implémenté avec une représentation de règles condition - action.
Un exécutif robuste gère l'exécution d'un plan tant qu'il reste valide et envoie des nouvelles requêtes au planificateur quand l'exécution échoue. L'intégration entre le planificateur et l'exécutif se résume donc à demander un plan au planificateur dans la situation actuelle, ou à recevoir du planificateur un nouveau plan.
Ces exécutifs sont chargés d'exécuter le plan au fur et à mesure du déroulement de la mission. Ils gèrent les interactions avec le planificateur et peuvent également gérer les interactions avec un module de diagnostic.
Ces langages permettent de modéliser un raffinement de tâches.

RAP

Le RAP [Firby, 89] (Reactive Action Packages) est un processus d'autonomie qui suit son but jusqu'à ce qu'il l'atteigne ou jusqu'à ce que toutes les possibilités pour l'atteindre soient épuisées. RAP permet l'exécution de tâches concurrentes et la gestion des exceptions. Dans le système RAP, la décomposition en sous tâches, la supervision et le traitement des exceptions sont organisés en unités discrètes. Un RAP peut se référer hiérarchiquement à d'autres RAPs. Son mécanisme d'exécution est décidé comme suit : à partir d'une file d'attente représentant l'ordre d'exécution des RAPs, l'ordonnancement est établit en fonction du modèle d'environnement et de la situation actuelle. Si elle se trouve en tête de la file d'exécution, une RAP est exécutée. RAP permet aux programmeurs de spécifier des objectifs, des chemins (ou politiques partielles) associés à ces objectifs et des conditions dans lesquelles le chemin a des chances d'être une solution. RAP facilite la gestion

des inévitables défaillances qui surviennent avec les systèmes robotiques réels. Le programmeur peut spécifier des procédures de détection pour toutes sortes de défaillances et fournir une procédure de gestion d'exception pour chacune d'entre elles. RAP est utilisé dans la couche exécutive pour gérer les imprévus qui ne nécessitent pas de replanification.

RPL

Le RPL [McDermott, 93] (Reactive Plan Language) est une extension du RAP qui est orientée vers la planification haut niveau et la proposition de plans alternatifs pour les situations critiques. Il propose au dessus de LISP des variables appelées `fluent`, dont le changement peut entraîner l'exécution de tâches par les constructions `whenever` ou `wait-for`. La construction `with-policy` permet l'exécution d'un tâche qu'à la condition qu'une autre tâche (éventuellement un simple test) n'échoue pas.

```
WITH-POLICY WHENEVER Batt-Level ≤ 46
CHARGE-BATTERIES
WITH-POLICY WHENEVER NEAR-DOOR(RmA-118)
SAY("hello")
DELIVER-MAIL
```

Figure 41 Plan de livraison du courrier dans un bureau en RPL

ESL

ESL [Gat, 97] (Executive Support Language) partage des caractéristiques avec des langages comme RAP et RPL. C'est une extension du common Lisp.
La gestion des evènements repose sur la notion de conscience de pannes. Cette approche considère que les différentes issues peuvent être facilement classées en tant que succès ou échec. Ce dernier nécessite une réponse adaptée. Pour cela différentes constructions permettent de définir des actions à exécuter en cas d'échec, d'autres sont employées pour découpler des états d'accomplissement et les méthodes pour réaliser ces états.
Les divers comportements de reprise sont appelés par la construction WITH-RECOVERY-PROCEDURES. S'ils ne sont pas possibles, la procédure de réinitialisation est appelée par la construction WITH-CLEANUP-PROCEDURE.
Un évènement peut être employé pour synchroniser des tâches concurrentes multiples dans ESL. Une tâche peut attendre l'occurrence de plusieurs évènements simultanément. Des tâches multiples peuvent également attendre le même évènement simultanément. Le langage ESL est le composant exécutif d'un agent dans l'architecture IDEA et de son prédécesseur Remote Agent [Pell & all, 98].

```
(defun recovery-demo-1 ()
(with-recovery-procedures
( (:widget-broken
(attempt-widget-fix :broken)
(retry))
(:widget-broken
(attempt-widget-fix
:severely-broken)
(retry))
(:widget-broken :retries 3
(attempt-widget-fix
:weird-state)
(retry)) )
(operate-widget)))
```

Figure 42 Exemple du langage ESL

PRS

PRS [Ingrand & all, 96] (Procedural Reasoning System) s'appuie sur une base de données mise à jour dynamiquement contenant l'état de l'environnement et celui du robot, une bibliothèque de plans et un arbre des tâches. Chaque plan de la bibliothèque décrit une séquence d'actions et de tests afin d'effectuer certaines tâches ou de réagir à telle situation. PRS ne planifie pas en combinant des actions mais en choisissant parmi des plans alternatifs dans une bibliothèque de plan la plus complète possible celui qu'il doit exécuter. L'arbre des taches est l'ensemble dynamique des tâches en cours d'exécution. L'interpréteur reçoit de nouveaux évènements et buts internes, choisit des plans appropriés à partir d'eux mais aussi des croyances du système. Il place les procédures

choisies dans l'arbre des tâches, choisit une tâche parmi les racines des arbres de tâche en cours et l'exécute. Ceci peut avoir pour conséquence une action primitive ou l'établissement d'un nouveau but. Dans PRS, les buts sont des descriptions d'un état désiré ainsi que les comportements nécessaires afin d'atteindre cet état. OpenPRS est un système basé sur la notion de buts. Chaque procédure écrite par l'utilisateur (OP) est chargée de réaliser un but, et est capable de demander la réalisation de sous-buts. C'est à la charge du noyau OpenPRS de gérer cet ensemble de buts de manière concurrente.

```
(defka |Long Range Displacement|
:invocation (achieve (position-robot $x $y $theta))
:context (and (test (position-robot
@current-x @current-y @current-theta))
(test (long-range-displacement
$x $y $theta @current-x @current-y
@current-theta)))
:body ((achieve (notify all-subsystems displacement))
(wait (V (robot-status ready-for-displacement)
(elapsed-time (time) 60)))
(if (test (robot-status ready-for-displacement))
(while (test (long-range-displacement
$x $y $theta @current-x
@current-y @current-theta))
(achieve (analyze-terrain))
(achieve (find-subgoal $x $y $theta
@sub-x @sub-y @sub-theta))
(achieve (find-trajectory $x $y $theta @sub-x
@sub-y @sub-theta @traj))
(& (achieve (execute-trajectory @traj))
(maintain (battery-level 0.200000)))
(test (position-robot @current-x @current-y
@current-theta)))
(achieve (position-robot $x $y $theta))
else
(achieve (failed)))))
```

Figure 43 Exemple de code PRS pour un déplacement sur une grande distance

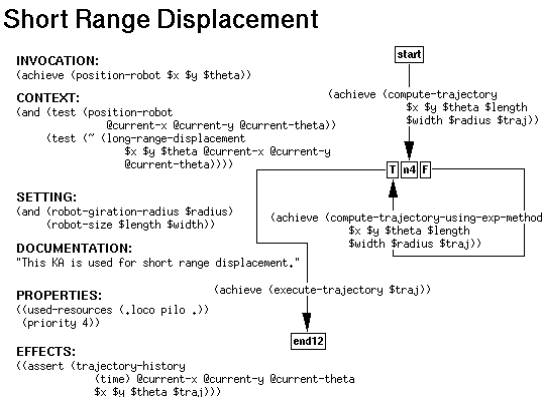


Figure 44 Exemple de code PRS pour un déplacement sur une petite distance

TDL [Simmons & all, 98] (Task Description Language) La représentation de base utilisée dans TDL est l'arbre des tâches où chaque nœud est lié à une action. Ces actions peuvent exécuter des calculs, ajouter dynamiquement des nœuds à l'arbre des tâches ou à exécuter une certaine action physique. Les actions liées aux nœuds utilisent les données des capteurs pour décider quels nœuds seront ajoutés à l'arbre et comment les paramétrer. TDL permet de décomposer une tache, de synchroniser des tâches secondaires et de surveiller l'exécution et les exceptions de traitement. TDL est un générateur de contrôleur : l'utilisateur écrit une spécification dans un dialecte proche du C++, qui est alors utilisé pour générer un contrôleur en C++.

<simple-constraint>:
expand first | **delay expansion**

<constraint>:
<constraint> , <constraint> | <simple-constraint>
| **sequential** <constraint-option> [<tag>]
| **serial** [<tag>] | **parallel** | **wait**
| **disable** [<constraint-option>] **until** <event>
| **disable** [<constraint-option>] **until** <absolute-time>
| **disable** [<constraint-option>]
for <relative-time> [**after** <event>]
| **terminate at** <event>
| **terminate at** <absolute-time>
| **terminate in** <relative-time> [**after** <event>]

<event>:
<tag> [<constraint-option>] <state>

<tag>:
<task name> | **self** | **parent** | **previous**

<constraint-option>:
handling | **expansion** | **execution**

<state>:
enabled | **active** | **completed**

Figure 45 Les contraintes de synchronisation dans TDL

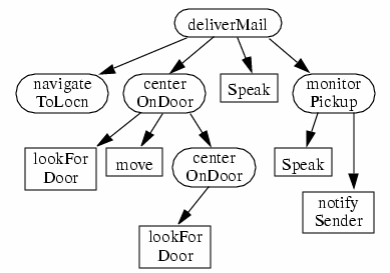


Figure 47 Exemple d'un arbre de tâches

```
Goal deliverMail (int room)
{
  double x, y;
  getRoomCoordinates(room, &x, &y);
  spawn navigateToLocn(x, y);
  spawn centerOnDoor(x, y)
      with sequential execution previous,
        terminate in 0:0:30.0;
  spawn speak("Xavier here with your mail")
      with sequential execution centerOnDoor,
        terminate at monitorPickup completed;
  spawn monitorPickup()
      with sequential execution centerOnDoor;
}

Goal centerOnDoor (double x, double y)
      delay expansion
{
  int whichSide;
  spawn lookForDoor(&whichSide) with wait;
  if (whichSide != 0) {
      if (whichSide < 0)
        spawn move(-10); // move left
      else
        spawn move(10); // move right
      spawn centerOnDoor(x, y)
        with disable execution until
          previous execution completed;
  }
}
```

Figure 46 Définition de deux tâches dans TDL

PILOT

Le langage graphique d'action impératif interprété **PILOT,** a été développé pour donner à l'utilisateur la possibilité de modifier en ligne une mission. L'aspect graphique est une de ses originalités.

PILOT [Le Rest, 96] (Programming and Interpreted Language Of actions for Telerobotics). Ce langage comprend quatre primitives : la séquence, la conditionnelle, l'itération, le parallélisme et la préemption. Basé sur une sémantique opérationnelle, il est interprété afin de permettre à l'opérateur humain d'intervenir dans les missions en ligne, et graphique pour que son utilisation soit plus aisée et plus visuelle. L'architecture logicielle, bâtie sur le modèle hiérarchisé, est composée de trois modules: un interpréteur, une base de connaissances et un module d'exécution. Le système décrit a été développé et a permis de programmer un robot mobile à roues. Ce langage est doté d'un système de contrôle dont les différents modules ont été modélisés à l'aide d'automates à états finis, mais dont la mise en oeuvre n'avait fait l'objet ni d'une génération automatique à partir des spécifications, ni d'une vérification formelle. [Nana & all, 03], suite à l'application de techniques de tests statiques et dynamiques aux programmes de l'interpréteur de plans PILOT dans un travail précédent, modélisent les algorithmes d'interprétation, vérifient leur conformité par rapport à la sémantique opérationnelle du langage, corrigent les éventuels dysfonctionnements, puis re-génèrent le code de l'interpréteur à partir du modèle validé.

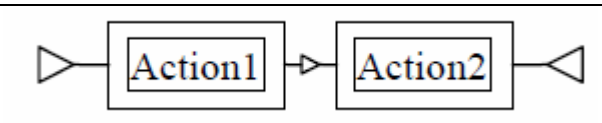


Figure 48 Exemple de séquence dans PILOT

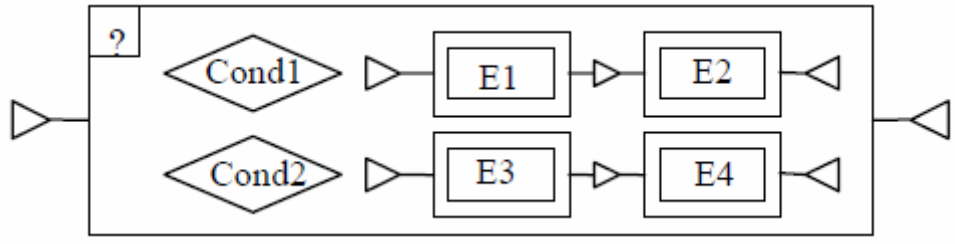


Figure 49 Exemple de condition dans PILOT

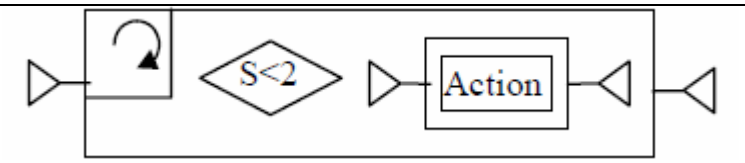


Figure 50 L'itération (conditionnelle) dans PILOT

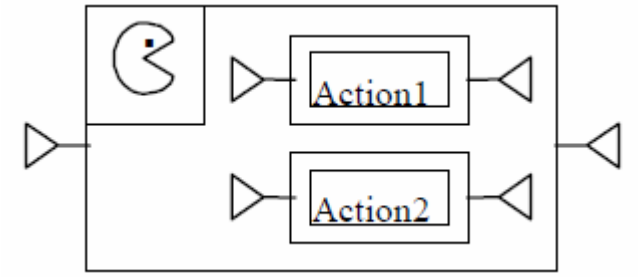


Figure 51 Préemption dans PILOT

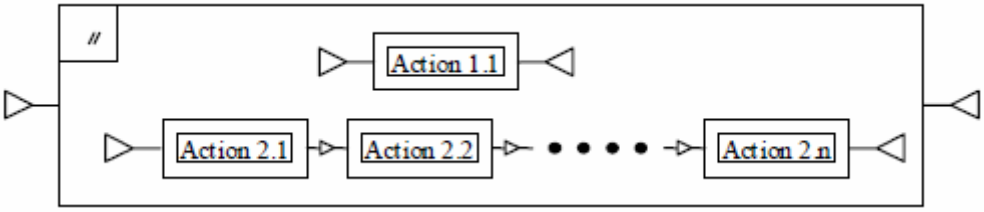


Figure 52 Exemple de plan avec parallélisme

Du point de vue des erreurs, leur représentation et leur gestion au niveau de l'exécutif robuste comprend :
1. la prise de décisions : ne jamais conduire le système dans une situation qui n'a pas été prévue. C'est le rôle des systèmes de planification.
2. la définition et la détection des erreurs, qu'elles viennent des couches de décision ou de la perception. C'est le rôle du diagnostic.
3. la récupération des erreurs qui ont été détectées mais qui ne sont pas prises en compte par le modèle utilisé par les planificateurs. C'est le rôle des systèmes dit de supervision : récupérer des situations non-nominales sans qu'elles aient été prises en compte par les outils de planification.

Le problème majeur entre ces trois points est la capacité à représenter les erreurs : la détection d'erreur n'est pas utile si le système de supervision n'est pas capable de la représenter.
D'une certaine manière la robustesse de ces exécutifs reposent sur des systèmes de surveillance de type « safety bags » dont les fonctions reconnues sont la vérification de la sécurité-innocuité des instructions générées par une commande, réaction à des événements extérieurs. Par exemple un des rôles de l'exécutif est de vérifier que les paramètres envoyés par la planification aux modules fonctionnels sont dans des plages plausibles.
Ainsi, à moins d'effectuer un recensement complet et non trivial des erreurs, il est impossible de prouver la complétude du « safety bag » surtout si des paramètres restent inconnus.

1.6.2 Langages de haut niveau pour systèmes hybrides

Un système hybride porte sur des domaines de valeurs discrets et des domaines de valeurs continues. Les valeurs des capteurs peuvent être considérées comme des variables qui varient avec le temps prenant leurs valeurs dans un domaine continu et des variables discrètes. En tant que système hybride, les comportements d'un robot ou d'un groupe de robot peuvent être spécifiés à l'aide d'équations différentielles. Les détails d'implémentation comme les méthodes de résolution des systèmes d'équations, le lien avec les capteurs, les actuateurs, ou les communications avec d'autres composants sont cachés au spécificateur de la mission. La définition formelle de ces langages permet de résonner sur les stratégies de contrôles et éventuellement de les changer. De plus, ils proposent un opérateur de composition de comportement, base de la réutilisabilité des contrôleurs. Dans un certain sens, ils sont des langages de prototypages.

Yampa

Yampa [Hudak & all, 02] utilise les combinateurs arrows [Hughes, 00], une généralisation des monads, pour représenter les principaux concepts FRP. Un constructeur de type d'arité deux fournissant trois opérations lui conférant des propriétés algébriques est un « arrow ». Doté d'un sucre syntaxique permettant de nommer les signaux, le FRP « arrowisé » réduit l'expression des programmes. La discipline qu'impose l'emploi des arrows qui interdisent une référence à un signal entier rend Yampa plus conforme aux contraintes temps réels. FROB [Peterson, 99] (Functional ROBotics) est une couche légère au dessus de FRP/Yampa, lui-même utilisant le langage Haskell pour le domaine de la robotique. Frob offre des interfaces prédéfinies pour les actionneurs et les capteurs, une infrastructure de communication, un simulateur, une bibliothèque pour la vision et des versions spécialisées de services fonctionnels comme les tâches. Yampa et Frob permettent au programmeur de spécifier les réactions du robot avec un plus petit effort de codage par rapport aux langages procéduraux. Frob permet de contrôler une équipe de la Robocup, comme GRL [Horswill, 00] (General Robotic Language). Ce dernier offre un exécutif Scheme plus léger pour l'embarqué. Pour améliorer la portabilité du concept de FRP, une extension permettant d'utiliser ce concept en C++ a par ailleurs été proposée dans [Dai & all, 02]. De plus, Dance [Huang & all, 03] est un langage FRP pour le contrôle de robots humanoïdes. Plutôt que de se focaliser sur les actionneurs et les capteurs, il propose des abstractions de mouvement de plus haut niveau.

```
import AFrob
import AFrobRobotSim
player1_1 :: SimbotController
player1_1 = rcStop
player1_2 :: SimbotController
player1_2 = rcGoToBall2 1.5
player1_3 :: SimbotController
player1_3 = rcStop
player2_1 :: SimbotController
player2_1 = rcStop
player2_2 :: SimbotController
player2_2 = rcGoToBall2 0.5
player2_3 :: SimbotController
player2_3 = rcStop

main = playSoccer [player1_1, player1_2, player1_3]
                  [player2_1, player2_2, player2_3]
```

Figure 53 Initialisation de l'équipe Robocup dans Yampa/FROB

```
-- Goes towards the ball.
RcGoToBall :: Velocity -> SimbotController
RcGoToBall vd rps = proc sbi -> do
    Let (phi, d) = head (aotBalls sbi ++ [(0.0,0.0)])
        h        = odometryHeading sbi
    rcHeadingAux rps -< (sbi, vd, h + phi)


rcGoToBall2 :: Velocity -> SimbotController
rcGoToBall2 vd rps =
    let loop = switch (rcGoToBall vd rps &&& rsStuck)
$ \_ ->
                switch (constant (mrFinalize (ddVelTR
(-vd) 0.3))
                        &&& after 2.5 ()) $ \_ ->
                loop
    in
        loop


-- Goes in the desired heading at the desired speed.
RcHeadingAux :: SimbotProperties ->
                SF (SimbotInput,Velocity,Heading)
SimbotOutput
RcHeadingAux rps =
    Let vMax = rpWSMax rps
        k    = 2
    in proc (sbi,vel,hd) -> do
         let vel' = lim vMax vel
         let phi = normalizeAngle (hd -
odometryHeading sbi)
         let vel'' = (1 - abs phi / pi) * vel'
         returnA -< mrFinalize (ddVelTR vel''
(k*phi))

lim m y = max (-m) (min m y)


-- Stops the robot.
rcStop :: SimbotController
rcStop _ = constant (mrFinalize ddBrake)
```

Figure 54 Les programmes de contrôle en Yampa/FROB

CHARON

CHARON [Alur & all, 00] (Coordinated control, Hierarchical design, Analysis, and Run-time mONitoring of hybrid systems) est un langage de specification modulaire pour systèmes hybrides et pour définir les stratégies de contrôle de robot. Les briques pour décrire l'architecture du système sont les agents qui communiquent avec leur environnement avec des variables partagées, les modes (comportements) et les contraintes (loi de commandes). Le langage permet la concurrence via la composition d'agents, la restriction du partage d'information, et l'instanciation pour permettre la réutilisation.
Les modes permettent de décrire le flot de contrôle d'un agent atomique. Un mode est simplement une machine hiérarchique à états finis qui peut avoir des sous modes connectés par des transitions. Une variable peut être déclarée localement dans un mode pour restreindre sa portée. Des points d'entrée et de sorties explicitement définis peuvent connecter deux modes. Des modes peuvent être partagés dans des contextes différents via l'instanciation. Les exceptions par des transitions de groupes depuis le point de sortie par défaut de tous les modes composants peuvent restaurer l'état local avant la dernière sortie.
Un agent peut aussi communiquer avec son environnement par des voies de communications. Des agents complexes sont bâtis à partir d'autres agents. Les agents atomiques ont un mode qui represente un flot de contrôle. Les modes peuvent avoir des sous modes et des transitions vers eux. Les modes représentent dans CHARON la hiérarchie des comportements pour la conception. Les compositions hiérarchique et séquentielle de mode sont possibles. Le comportement de chaque agent atomique est décrit par un simple mode de haut niveau qui correspond à un simple thread. Chaque mode a une interface de données bien définie pour le partage d'informations par des variables globales et une interface de contrôle consistant à la liste des points d'entrée et de sortie. Le mode de haut niveau qui est activé après l'instanciation de l'agent n'est jamais désactivé et possède le point d'entrée init. Les feuilles de la hiérarchie de contrôle (le modes atomiques) sont des comportements purement continus qui spécifient les lois de contrôles (équations différentielles, contraintes algébriques sur des valeurs de variables). De plus un invariant peut être précisé pour définir les conditions d'activité du mode. Dès que l'invariant est violé, le mode doit être quitté en utilisant une des transitions prévues.

```
agent base() {mode topBase=topBaseMode()}
agentvmanipulator(){mode
topManipulator=topManipulatorMode()}

mode topBase() {
...
channel of real myState
channel of real otherState
receive (otherState, y)
send (myState, x)
diff dynamicsf{d(x) == fb (x, y) }
...
}
mode topManipulator() {
...
channel of real myState
channel of real otherState
receive (otherState, x)
send (myState, y)
diff dynamicsd(y) == fm (x, y) ;
...
}
agent Robot() {
private channel of real baseToManipulator,
manipulatorToBase;
agent theBase = base () [baseToManipulator,
manipulatorToBase]
agent theManipulator = manipulator ()
[baseToManipulator, manipulatorToBase]
}
```

Figure 55 Définition d'un agent composite dans CHARON.

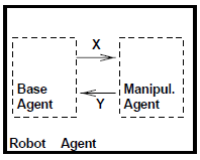


Figure 56 Composition parallèle de l'agent Base et de l'agent Manipulator

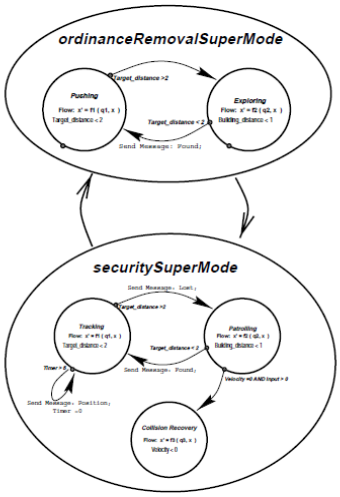


Figure 57 Composition hiérarchique de modes pour former un mode de plus haut niveau.

CCL

Le langage CCL [Klavins, 03], [Waydo & all, 03] (Computation and Control Language) basé sur des commandes gardées afin exprimer des systèmes dont contrôles et calculs sont entremélés. Un programme CCL est un ensemble de commandes gardées qui peuvent mettre à jour des variables continues ou discrètes et où le résonnement utilise une logique temporelle. L'application proposée est un système de robots jouant un jeu de type capture-the-flag (des équipes d'adversaires doivent capturer les drapeaux des autres). Le langage formel permet de vérifier certaines propriétés. Puis l'auteur propose CCLi [Klavins, 04] (CCL interpreter) où il donne l'exemple de vol en bande (flocking). Le contrôle de robots est simplement la composition parallèle de programmes CCL.

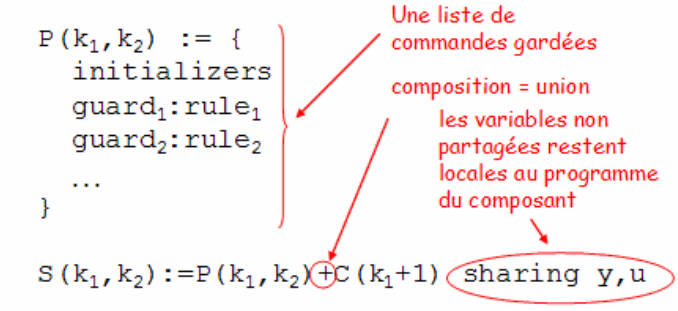


Figure 58 Un petit exemple de programme CCL

```
Program main() := compose i in range n :
robot ( Xlist, alist, i, n, r, d1, d2 )

program robot ( X0, a0, i, n, r, d1, d2 ) :=
receiver ( X0, a0, i, n, r )
+ mover ( d1 ) sharing X, a, N, t
+ sender ( i, n, d2 ) sharing X, a;
```

Dans cet exemple de programme CCLi, chaque robot reçoit une information provenant que de ces voisins. Il va alors faire une moyenne des informations et va l'utiliser pour son propre contrôle.

```
program sender ( i, n, delta ) := {
needs X, a;
  t := dclock();
  j := 0;
  dclock() - t > delta : {
    send ( [ to := j, from := i,
    a := a, X := X ] ),
    t := dclock(),
    j := ( j + 1 ) mod n
  }
}

program mover ( delta ) := {
  needs X, a, N, t;
  dclock() - t >= delta : {
    a := update_heading a N,
    X := update_pos delta X a,
    N := smult 0.0 N,
    T := dclock()
  }
}

program receiver ( X0, a0, i, n, r ) := {
  X := X0;
  a := a0;
  N := makelist n 0.0;
  inbox ( i ) : {
     N := recv_filter r i X N
  }
}
```

La fonction externe `dclock()` donne le millième de seconde le nombre de seconde passée depuis le début de la mission.

La fonction `recv_filter()` qui n'est pas présentée filtre les informations en ne retenant que les celles provenant que des voisins du robot.
La commande gardée vérifie si un message est disponible. Si c'est le cas, le robot met à jour le vecteur de voisins.

Les fonctions `update_heading()` et `update_pos()` encodent en CCLi les équations suivantes :

$$\theta_i(k+1) = \frac{1}{|N_i(k)|} \sum_{j \in N_i} \theta_j(k) \qquad (1)$$

où

$$N_i(k) \triangleq \{j \mid ||X_i(k) - X_j(k)|| \leq r\}$$

et

$$X_i(k+1) := X_i(k) + \delta(\cos\theta_i(k), \sin\theta_i(k)).$$

Figure 59 Exemple de programme CCLi de vol en bande

1.6.3 Langages de contrôle spécifiquement orientés agents

GOLOG

GOLOG (alGOl in LOGic) [Levesque & all, 97] est un langage de programmation logique pour des domaines dynamiques basé sur le calcul de situation qui permet de raisonner sur le monde et sur ses évolutions (tous les mondes pouvant résulter d'une ou de plusieurs actions de l'agent). Les programmes en GOLOG spécifiant le comportement d'un agent peuvent être écrits avec un niveau élevé d'abstraction : les actions complexes d'un agent se décomposent en actions primitives qui font référence normalement à des actions externes aux agents comme par exemple ramasser un objet, par opposition à des actions qui n'affectent que leur état interne. Le modèle de l'environnement est mis à jour dynamiquement en fonction des règles définies par le programmeur. Ainsi le système analyse les impacts des différentes actions avant d'en choisir une à exécuter. Outre la spécification d'un programme de contrôle avec des possibles actions non déterministes, le programmeur doit aussi fournir un modèle complet du robot et de son environnement. Quand le programme de contrôle atteint un point de choix non déterministe, il invoque un planificateur (sous la forme d'un démonstrateur de théorèmes) pour déterminer l'action suivante. On peut alors spécifier des contrôleurs partiels et s'appuyer sur des planificateurs prédéfinis pour concevoir un contrôle de choix final. Ainsi l'interpreteur GOLOG maintient automatiquement le modèle d'environnement pour l'agent et il a la responsabilité du choix de la séquence d'actions légales à transmettre au module d'exécution d'actions primitives. Certaines limitations de GOLOG ont été levées par pGOLOG (version probabiliste qui permet de gérer les états initiaux incomplets), ccGOLOG (gestions des processus continus et concurrents) et GOLEX. Golex veut combler le fossé entre les techniques symboliques de haut niveau avec le contrôle de bas niveau d'un robot. Golex permet d'utiliser GOLOG avec le logiciel de contrôle distribué RHINO en fournissant un niveau d'abstraction élevé, un moniteur d'exécution et la perception. ICPGOLOG [Dylla & all, 03] combine tous ces aspects dans un même langage et a été expérimenté au niveau de la ligue de simulation mais aussi de celle de moyenne taille de la Robocup. ICPGOLOG offre les constructions comme la séquence d'actions, `pconc` (pour des actions en concurrence), `prob` (actions probabilisée), `waitFor` (interruption par évènement), `if`, `while`, `proc` (procédures récursives), un opérateur de choix non déterministe, applicable lors de l'exécution et une condition de test qui évalue une expression du premier-ordre.

```
withPol(whenever(battLevel ≤ 46,
seq(grabWhls, chargeBatteries, releaseWhls)),
withPol(whenever(nearDoorA-118,
seq(say(hello), waitFor(¬nearDoorA-118)))
withCtrl(wheels, deliverMail)))
```

Figure 60 L'exemple donné avec RPL traduit en ccGOLOG

```
proc (try direct pass(Own, TargetPlayer),
  [ directPass(Own, TargetPlayer, pass NORMAL),
    pconc (prob_intercept_direct pass(closestOpp-
       ToPass(TargetPlayer), TargetPlayer),
    waitFor (or(ball_near_player(TargetPlayer),
        ball_far(TargetPlayer))))]).
proc (prob_intercept_direct_pass(Opp, TargetPlayer),
prob(0.7, intercept direct pass1(Opp, TargetPlayer),
intercept_direct_pass2(Opp, TargetPlayer))).
```

Figure 61 Avec ICPGOLOG, projeter une double passe avec le comportement d'un opposant probabilisé

MRL

MRL [Nishiyama & all, 98] (Multiagent Robot Language) est un langage multi-agent pour le contrôle de robot Comme TAPIR, il y a une hiérarchie d'agents sous forme d'arbre (Figure 62), l'agent racine contrôlant tous les autres agents. Les feuilles sont les capteurs et les actionneurs. MRL considère les robots et les capteurs comme des agents intelligents et s'intéresse à la communication au niveau sémantique entre les agents. Un agent peut être constitué de sous agents. Par contre un sous agent n'a qu'un seul super agent. Chaque agent possède sa propre base de connaissance, de règles et de procédures afin d'exécuter les taches demandées par un agent externe. Un agent a deux voix de communication, une vers ses sous agents et l'autre vers son super agent. Le super agent contrôle ses sous agents par diffusion de messages et joue le rôle de routeur pour la communication entre chacun de ses composants. Des variables synchronisation peuvent être utilisée. La Figure 64 montre le squelette de définition d'un agent.
MRL permet la gestion des conflits d'accès, la manipulation d'évènements et la négociation entre agents. Le langage MRL a été testé avec un système multi-robot constitué de quatre manipulateurs, d'un capteur de vision et d'une caméra mobile (Figure 63). MRL a aussi été appliqué aux robots mobiles avec succès. Cependant MRL est peu lisible.

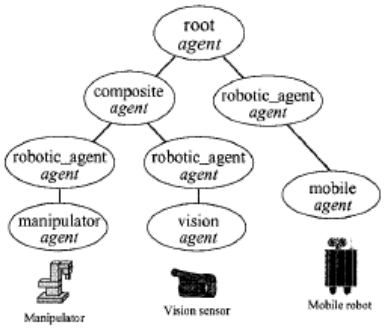


Figure 62 La hiérarchie d'agents dans MRL

```
:- agent(<agent_name>).
 new(·) :-                    % Agent instantiation
   init(·), run(·).

init(·) :-                    % Instantiating subagents
   …, #<subagent_name>:new(·), …

 run(·) :-                    % Behavior definition for express
   Handling express message
alternatively.
 run(·) :-                    % Behavior definition for normal case
   Headling normal message
```

Figure 64 Syntaxe de définition d'un agent MRL

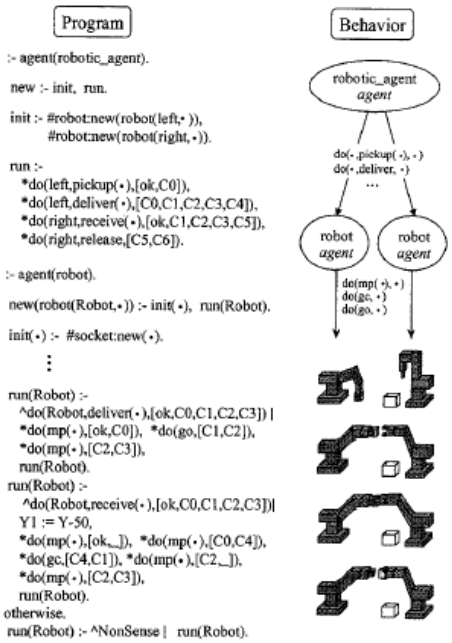


Figure 63 Un programme coopératif MRL utilisant une synchronisation

TAPIR

Le langage d'actions TAPIR [King, 02] étend HAC (cf. 1.4.2) et permet de décrire les comportements et les tâches dirigées par les buts. C'est un langage générique de planification pour le contrôle d'agents semi

déclaratifs implémentés en Common Lisp. Il définit un modèle de processus, un langage de description de ressource et des opérateurs de contrôle. La spécification est basée sur les actions qui nécessitent des ressources qui peuvent être des parties d'agents, des agents, des groupes d'agents. La planification consommant beaucoup de ressources, son exécution est répartie sur plusieurs cycles d'horloges. Des buts latents (qui deviennent actifs selon certaines conditions) sont pris en compte par le gestionnaire de plan pour obtenir la réactivité par rapports à des évènements imprévus.

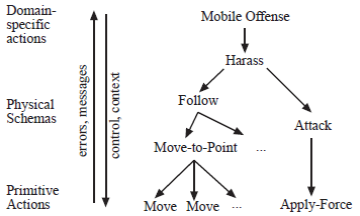


Figure 65 Une hiérarchie d'action dans TAPIR

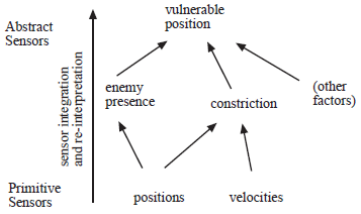


Figure 67 Une hiérarchie de capteurs dans TAPIR

```
(defection move-to-random-point ()
:documentation "Move an agent to a randomly
selected location."
:resources (agent)
:do
(move-to-point
:target-geom (find-random-location-for-agent
the-simulation agent))
:on-message (message
(stop-children)
:restart))
;;; -----------------------------------------------
(defaction swarm ()
:documentation "Move any number of agents to
random locations repeatedly."
:resources ((agents :count :all))
:do
(:foreach agent in agents :in-parallel
(move-to-random-point :agent agent))
:ignore-messages)
```

Figure 66 Implémentation d'un comportement en essaim multi agents avec TAPIR

CDL

CDL [MacKenzie & all, 97] (Configuration Description Language) est un langage de programmation basé agent fondé sur la théorie d'agent sociétal qui présente une vue uniforme de tous les composants d'une configuration robotique comme des agents abstraits. Il utilise la notation fonctionnelle pour permettre la construction récursive d'assemblages qui une fois nommés seront la base d'autres assemblages. Ce langage est le format de sauvegarde des configurations définit à l'aide de l'éditeur graphique CfgEdit (Figure 68) manipulant des opérateurs élémentaires. La représentation graphique correspond à un programme CDL. La mission multi-robot est ensuite compilée au travers d'une série de langages pour arriver à un programme exécutable par un robot particulier, par exemple un robot Pioneer. Notamment CDL est traduit pour une architecture AuRA en CNL (Configuration Network Language) puis en C++ puis enfin en langage machine.

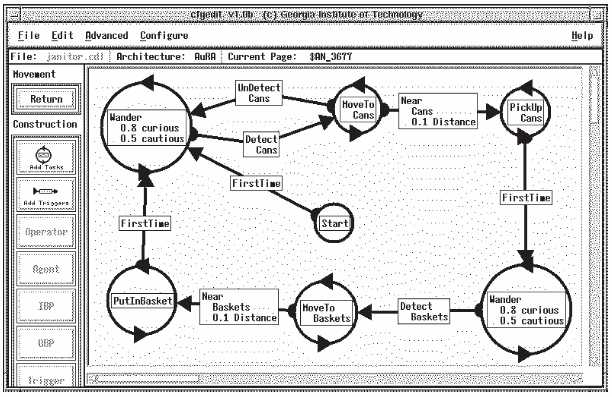
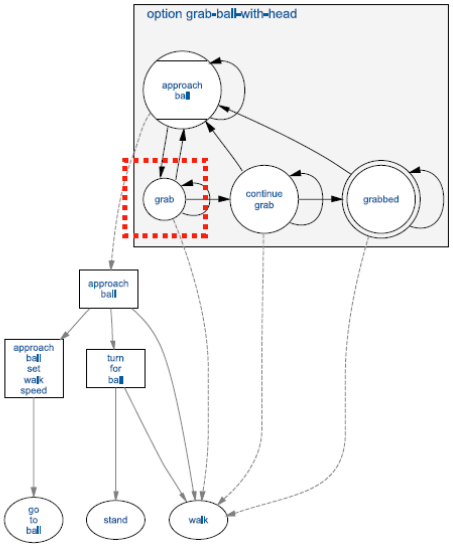


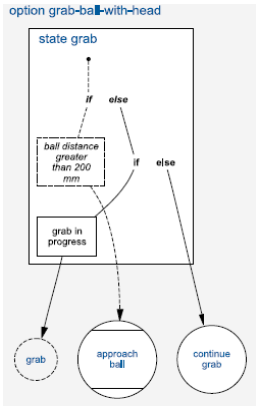
Figure 68 L'éditeur de configuration de mission. Le graphique est traduit en CDL pour le stockage de la configuration

XABSL

XABSL [Lötzsch, 04] (eXtensible Agent Behavior Specification Language) est un langage basé sur le langage XML permettant de développer des comportements pour agents autonomes. Un agent y est considéré comme étant un regroupement hiérarchique de modules de comportements appelés options. La prise de décision au sein des options se fait grâce à des machines à états. La hiérarchie peut se représenter graphiquement et est toujours terminée par des basic behaviors. Un exemple d'option développée pour l'équipe GermanTeam est présenté à la Figure 69 puis une transition d'état propre à cette option est présentée.
Le langage XML permet de se détacher complètement de l'implantation matérielle du système robotisé. Les comportements de plus bas niveaux peuvent alors être programmés dans divers langages, ce qui rend l'architecture modulaire et générique. L'utilisation des machines à états pour la prise de décision à l'aide d'outils graphiques, si elle facilite le développement et la documentation du système décisionnel, peut cependant être contraignante pour l'implantation de mécanismes de décision arbitraires.
L'architecture proposée avec XABSL a été développée et utilisée depuis quelques années par l'équipe GermanTeam de la « Sony 4-Legged Robot League ». En utilisant cette architecture, cette équipe a remporté les éditions 2004 et 2005 de la Coupe du Monde. Il faut noter que XABSL ne permet pas d'ajouter dynamiquement un agent à un groupe. De plus l'emploi de XML ne permet pas d'interfacer aisément les agents existants.



Une machine d'état d'option. Les cercles représentent les états. Un cercle avec deux lignes horizontales représente l'état initial. L'état cible correspond à un double cercle. Une flèche entre deux état indique qu'il y a au moins une transition. Une flèche en pointillés indique les options (rectangles) ou les comportements primitifs (ellipses) qui deviendront actifs lorsque l'état correspondant est actif.



Un arbre de décision d'état. Les feuilles sont des transitions vers d'autres états. Le cercle en pointillé dénote une transition vers le même état.

```
if ((ball.time-since-last-seen-consecutively < 200)    // ball distance greater than 200 mm
    && (ball.consecutively-seen-time > 100)
    && (ball.seen.distance > 200)
    && (ball.seen.distance < 800)                  )
{
  transition-to-state(approach-ball);
}
else
{
  if (time-of-state-execution < 1000)   // grab in progress
  {
    transition-to-state(grab);
  }
  else
  {
    transition-to-state(continue-grab);
  }
}
```

Pseudo code de l'arbre de décision de l'état « grab »

Figure 69 XABSL pour une application robots footballeurs.

A noter qu'un autre langage XML centrée sur l'interaction hommes et multi-robot via les protocoles web, nommé RoboML [Makatchev, 00] (Robotic Markup Language) se contente d'encapsuler le programme de contrôle des robots dans une balise XML.

HoRoCoL

HoRoCoL [Dubois & all, 03], [Duhaut & all, 06], [Duhaut & all, 06b], [Le Guyadec & all, 05], [Le Guyadec & all, 05b] (Homogeneous Robotic Component Language) qui a été developé pour le projet `MAAM` [Duhaut, 02] s'appuie sur l'API du robot MAAM dont l'étude préalable était nécessaire afin de connaître le type d'objet informatique manipulé par le langage. Chaque molécule offre une vision synchrone et centralisée au programmeur (point de vue macroscopique). Le mécanisme de compilation permet de passer de cette vision à une exécution asynchrone et répartie (point de vue microscopique). Par ailleurs, un modèle où le même code peut s'exécuter sur des structures aux propriétés topologiques identiques mais de tailles différentes a été retenu. Le langage HoRoCol est construit au dessus d'un noyau classique de langage séquentiel (variables scalaires, affectation, conditionnelle, boucle). La partie vectorielle du langage permet la manipulation concurrente d'atomes. A cette fin, des structures de contrôle spécifiques (`where`, loop parallèle, `;`, `parofseq`, `seqofpar`) sont proposées. Le code s'exécute implicitement sur l'ensemble des atomes actifs. Lorsque du code doit être appliqué à un sous ensemble de la molécule, il suffit d'activer la sous partie souhaitée via la construction `where`. Contrairement à un if séquentiel qui n'exécute qu'une de ses branches, le `where` permet l'exécution concurrente de 2 programmes s'appliquant à des sous ensembles distincts de la molécule. De plus le `where` ne termine que lorsque tous les atomes ont terminé l'exécution de leur branche. On a donc une synchronisation implicite en fin de `where`.

```
import MAAM.xml;                                // declaration file of the maam agent type
import Clock.xml;                               // declaration file of the clock agent type

programHorocol walking_row_with_timeout {       // the goal for the row is to walk during 3 minutes
 type maam use MAAM.xml;                        // association of the maam agent type with its xml interface
 type clock use Clock.xml;                      // association of the maam agent type with its xml interface
 maam a1, a2, a3 = newAgent(maam);              // instanciation of 3 agents with the maam type
 clock watch=newAgent(clock);                   // instanciation of 1 agent with the clock type
 ...                                            // construction of the initial state of each agent :
 ...                                            // - a row of 5 interconnected atoms with all legs centered;
 ...                                            // - a clock with the current time.
 ||(°p_row°,[p_clock]);                         // run in parallel 2 control programs
}
```

Figure 70 le programme HoRoCol au niveau social

Le programme p_clock ordonne à l'agent watch de faire un compte à rebours pendant 3 minutes. Le fichier `Clock.xml` définit la méthode `waitOneSeconde()` pour les agents de type `Clock`. Au bout des 3 minutes, le programme `p_clock` se termine, ce qui provoque la terminaison du programme `p_row` qui a été lancé en mode interruptible.

```
int time=180;                                   // time is a shared variable for clock typed agents
parofseq(clock) {
    where (true) {
        while (time>0){
            waitOneSecond();                    // send message to agent watch
            time--;
        }
    }
}
```

Figure 71 Le programme p_clock de l'agent logiciel clock

L'algorithme itératif présenté dans le programme `p_row` décrit le calcul permettant à cette dernière d'avancer (1 itération = 1 pas pour l'ensemble des atomes). Son invariant est qu'à chaque itération, les pattes se mettent d'abord en position arrière. Il comporte 3 phases qui peuvent se répéter indéfiniment :
- la première étape consiste à positionner la rangée toutes pattes en arrière (voir Figure 72.b);
- la deuxième étape consiste à lever puis à poser les pattes gauches en avant pour les atomes impairs et lever puis poser les pattes droites en avant pour les atomes pairs (voir Figure 72.c);
- la dernière étape permet de lever puis poser les pattes restantes en avant (voir Figure 72.d).

Pour la première étape, il n'est pas nécessaire de lever les pattes. Aucun synchronisme entre pattes est exigé : on utilisera la construction `parofseq`. Afin de maintenir la rangée en équilibre, il est nécessaire de ne lever au plus qu'une patte sur 2 par atome à un instant donné. Les deux dernières phases exigent un synchronisme entre agents. On a donc une séquence de 2 `seqofpar`. La Figure 72.b montre l'état de la rangée avant le premier bloc seqofpar. La figure 3.c montre l'état de la rangée avant le deuxième bloc seqofpar.

Lorsque le but est atteint (ici marcher pendant 3 minutes), le programme sort de la boucle infinie while portant sur les atomes de la rangée.

On voit que cet algorithme est adapté à des rangées de taille quelconque.

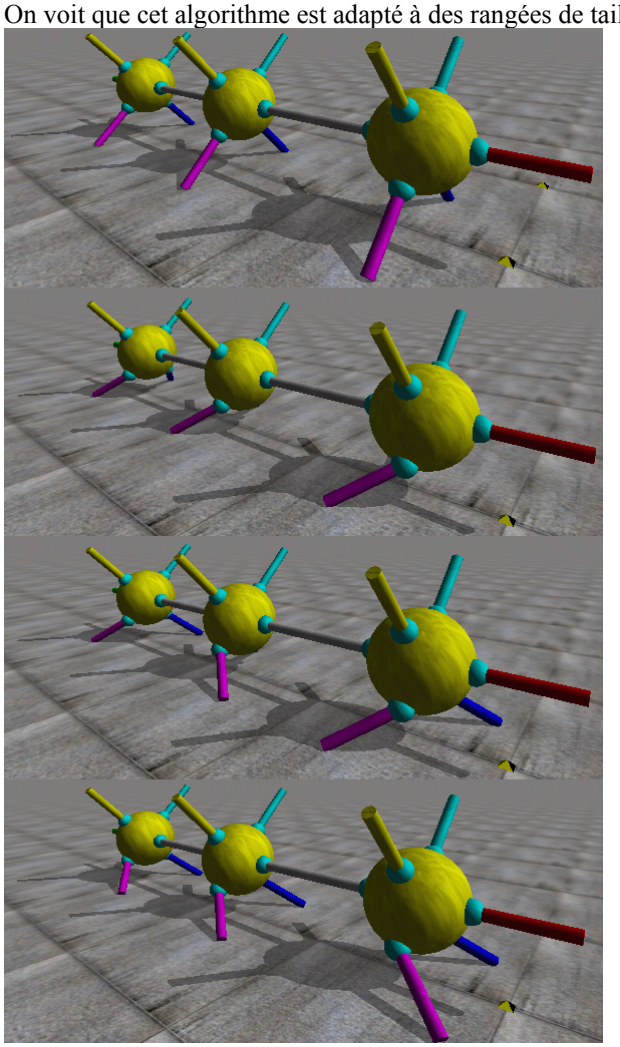


Figure 72 Les quatre phases du programme HoRoCoL

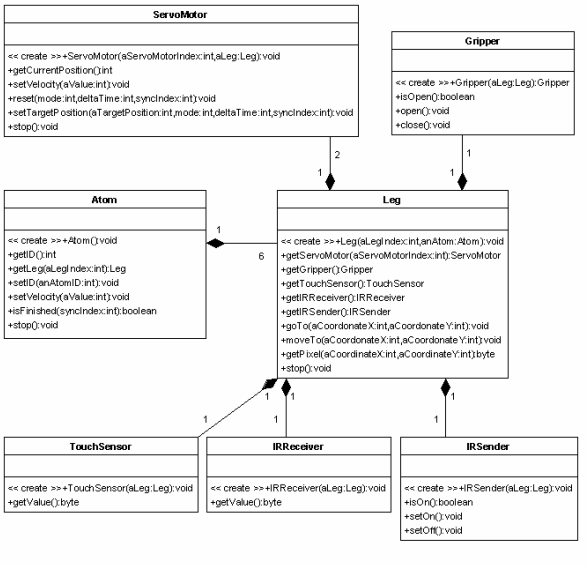


Figure 73 API des atomes robotiques

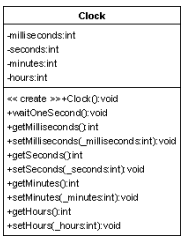


Figure 74 API de l'agent clock

Le code du programme `p_row` est le suivant :

```
while (true) {
    parofseq(maam){
        // put all legs on ground in back position
        where (true) {
            .getLeg(left).moveTo(min,min);
            .getLeg(right).moveTo(min,min);
        }
    }

    // the synchronized walk can start when all the agents have their legs on the ground behind
     seqofpar(maam){
        // lift, put at 45° ahead and descend the left legs for the odd atoms.
        where (maam.getID() % 2==1) {
            .getLeg(left).moveTo(min,middle);
            .getLeg(left).moveTo(max,middle);
            .getLeg(left).moveTo(max,min);
        };

        // lift, put at 45° ahead and descend the right legs for the even atoms.
        where (maam.getID() % 2==0) {
            .getLeg(right).moveTo(min,middle);
            .getLeg(right).moveTo(max,middle);
            .getLeg(right).moveTo(max,min);
        };
    }
```

```
    // All the legs on ground have to be in front

    seqofpar(maam){
        // lift, put at 45° ahead and descend the right legs for the odd atoms.
        where (maam.getID() % 2==1) {
            .getLeg(right).moveTo(min,middle);
            .getLeg(right).moveTo(max,middle);
            .getLeg(right).moveTo(max,min);
        };
        // lift, put at 45° ahead and descend the left legs for the even atoms.
        where (maam.getID() % 2==0) {
            .getLeg(left).moveTo(min,middle);
            .getLeg(left).moveTo(max,middle);
            .getLeg(left).moveTo(max,min);
        };
    }
}
```

Figure 75 le programme HoRoCoL p_row d'une rangée d'atomes robotiques.

Le langage HOROCOL a été abandonné pour MASL. En effet, nous avons découvert que la lourdeur de l'écriture seqofpar et parofseq pouvait avantageusement se simplifier en introduisant la notion de « porte » imbriquable alors que seqofpar et parofseq ne le sont pas.

Ceci termine l'exposé des langages existant de contrôle multi-robots. De la classification proposée, on retient que :

- les langages intermédiaires sont focalisés sur la bonne exécution des plans plutôt multi-robots mais en devant combiner des modules à forte disparité de représentation, le traitement des erreurs n'est pas complet ;
- les langages hybrides permettent le raisonnement sur les stratégies de contrôles multi-robots. Souvent couplets à des simulateurs, ils peuvent être vus comme une extension de MATLAB. Le portage sur des robots réels est dans un second temps et peut être considéré comme optionnel.
- Les langages de contrôles multi agents est notre cœur de cible donc nécessitent une étude par rapport aux propriétés définies dans l'introduction. Le tableau comparatif se trouve page 61 (Tableau 2 Langages orientés agents robotiques et les six propriétés).

1.7 Programmation orientée multi-agents

La programmation orientée multi agents diffère de la programmation orientée agents ou acteurs dans la sens où elle se focalise sur les interactions entre agents plutôt que sur la définition de leurs qualités intrinsèques. MASL en proposant des constructions sur des groupes d'agents permet une approche multi agents de la programmation.

1.7.1 Programmation orientée objets, programmation orientée agents et programmation orientée multi-agents

Les langages d'agents mettent à disposition du programmeur des abstractions de haut niveau qui facilitent l'expression du calcul de la mission. La distinction entre un langage d'agent et une architecture d'agent peut apparaître artificielle dans bien des travaux cites dans [Wooldridge & all, 95] qui ne sont pas focalisés sur les aspects spécifiques de la programmation robotique. Elle est cependant claire pour l'AOP [Shoham, 93] (la programmation orientée agent) : Agent0 [Shoham, 91], AgentSpeak [Weerasooriya & all, 95], 3-APL [Ross & all, 05]. Dans ces langages, les notions mentales qui caractérisent les agents apparaissent dans le langage lui-même, et la sémantique du langage est intimement liée à la sémantique de ces notions mentales. La programmation orientée agents peut être vue comme une spécialisation de celle orientée objets parce que les composants du programme sont maintenant des agents. Un compilateur d'agents permet de passer de spécifications de haut niveau au déploiement avec un effort réduit.
Une comparaison de la programmation orientée agent avec celle orientée objets, est proposée dans [Odell, 02]. Dans la programmation monolithique, les unités de logiciel représentent le programme dans sa globalité, entièrement spécifié par le programmeur et contrôlé par le système. La programmation modulaire répond à l'augmentation en terme de complexité, de besoins de mémoire des applications, ainsi qu'au besoin de réutilisabilité. Par contre l'état des unités (modules) restait toujours sous contrôle externe par instructions d'appel de procédures et fonctions. Dans la programmation orientée objets, l'état des objets est encapsulé grâce à des

méthodes d'accès ou d'instanciation. Par contre en programmation orientée objet classique, les objets sont considérés comme passifs et donc soumis au contrôle par envoi de messages et appel de méthodes. Enfin, afin de répondre aux besoins des agents, la programmation orientée agent se propose de concevoir les agents comme des "objets actifs avec initiative" – un peu comme les acteurs (*cf.* 1.2.2) – qui ont leur propre "thread" de contrôle et sont mus par des règles, des buts ou des intentions.

Tableau 1 L'évolution des approches de programmation : de la programmation monolythique à la programmation objet [Odell, 02]

	Programmation monolythique	Programmation structurée/modulaire	Programmation orientée objet	Programmation orientée agent
Comment une unité se comporte-t-elle ? (code de l'unité)	Non modulaire	Modulaire	Modulaire	Modulaire
Que fait une unité en cours d'exécution ? (état de l'unité)	Externe	Externe	Interne	Interne
Quand une unité s'exécute-t-elle ? (appel de l'unité)	Externe	Externe (appel)	Externe (message)	Interne (régles, buts)

A titre d'exemple, on peut citer RIDL [Verhaeghe & all, 05] (Robot-Intelligence Definition Language) qui propose de considérer un nouveau type de données dans le langage java, en plus des classes, les agents. Les agents sont plus que des objets : ils sont autonomes, ont des capacités de communication et de perception et ils opèrent dans un environnement non déterministe. On peut simuler un agent à l'aide d'un objet mais cela demande beaucoup de travail. Un nouveau type de données Agent est apparu comme extension du langage java. Il faut lui spécifier ses buts, ses capteurs et son comportement. Le compilateur d'agent peut optimiser le code. Notamment par l'assignement dynamique de Thread (pool de thread pour les agents actifs) et la définition de la priorité des threads. En analysant les dépendances entre les capteurs, les buts et les comportements, le compilateur est capable d'en déduire ces priorités automatiquement alors que c'est une tache la plus difficile lors de la programmation d'un système parallèle. RIDL permet de se focaliser sur la définition des comportements des agents. Ce langage, basé sur une idée intéressante, a été proposé par un ingénieur mais l'entreprise qui l'a commercialisé a fermé. Il ne subsiste que sous forme de brevet.

```
package ElectricityPlant ;
public agent ControlRoom {
  public int shortage = 0 : sensor;
  private MeasuredElectricity measured;
  private ElectricityDemand demand;

  private void calculateShortage() : behaviour
  {
     when (measured.amount.changes || demand.amount.changes) {
       setShortage(demande.amount-measured.amount.getElectricity());
      }
  }
  private void setShortage(int p_shortage) : goal
  {
    shortage=p_shortage;
  }
  public ControlRoom(MeasuredElectricity p_measured, ElectricityDemand p_demand) {
    measured=p_measured;
    demand=p_demand;
  }
}
```

Figure 76 Exemple de définition d'un agent dans RIDL

La notion de programmation orientée multi-agents

La perspective d'une Programmation Orientée Multi-Agents consiste à envisager un contexte dans lequel le programmeur n'aurait, pour un domaine d'application donné, pour un problème à résoudre ou un système à simuler, qu'à choisir les architectures d'agents, la représentation de l'environnement, les méthodes d'interaction, le type d'organisation, une dynamique particulière en termes d'émergence et de récursion, et à ensuite laisser la plate-forme se charger de construire le SMA résultant, directement connecté au système distribué cible.

1.7.2 Méthodes, modèles et programmation orientées multi-agents

Les méthodes proposées ici à titre d'illustration ne sont pas dépendantes d'un langage utilisé. MASL n'est pas contradictoire avec ces méthodes. Bien au contraire, les méthodes exposées sont en amont de MASL et leur utilisation permettra une « bonne programmation » MASL.
Ces études sur l'analyse, la conception, l'implantation et le déploiement de systèmes multi-agents/multirobots, bien que pour chacune d'entre elle à l'exclusion de la première une plateforme de développement multi-agents adhoc ait été développé, sont à rapprocher des efforts de standardisations des interfaces de contrôles des robots, Pyro et URBI. Pyro [Blank & all, 06] (PYthon RObotics) est une compilation d'interfaces open-source de robots pilotables à travers des scripts python. Beaucoup plus ambitieux, *URBI* [Baillie & all, 06] (Universal Robotic Body Interface) est un langage de script conçu pour fonctionner selon un mode client/serveur dans le but de contrôler un robot, ou plus largement, tous les types d'appareils disposant de moteurs et de capteurs. Ce sont les concepteurs de robots (éventuellement propriétaires) qui ont à fournir les pilotes permettant leur contrôle via URBI. Enfin Microsoft Robotics Studio est un environnement de développement intégré d'application robotique qui propose de nombreuses interfaces de contrôle de robots actuels.
Si l'étude de chaque méthode se fait sous un angle général, les notions de rôles, de groupes et de changement dynamique de rôles/groupes seront abordées car elles sont à la base de la programmation MASL. De plus nous n'abordons pas la notion d'apprentissage. Les méthodes citées n'ont pas forcément pour cibles les SMA adaptatifs. Pour des études plus poussées, voir [Amiguet, 00], [Ricordel, 01], [Savall, 03], [Seq, 03].

1.7.2.1 La méthode Cassiopee

Cassiopee [Drogoul & all, 98] repose sur quelques concepts clés : ceux de rôle, d'agent, de dépendances et de groupes. Son idée principale est qu'un agent n'est rien d'autre qu'un ensemble de rôles, parmi lesquels on distingue trois niveaux :
- les rôles individuels qui sont les différents comportements que les agents sont capables de tenir sans se soucier de la stratégie choisie pour les tenir ;
- les rôles relationnels qui concernent comment ils choisissent d'interagir avec un autre (en activant ou en désactivant les rôles individuels) avec le respect des dépendances mutuelles de leurs rôles individuels ;
- les rôles organisationnels ou comment les agents peuvent gérer leurs interactions pour devenir ou rester organisés (en activant ou en désactivant des rôles relationnels).

Il convient de noter que Cassiopee a représenté l'une des premières tentatives de description générique des étapes de conception d'un SMA. Nombre de ses concepts se retrouvent maintenant dans la plupart des méthodes « orientées agent » disponibles dans la littérature, sous des formes un peu différentes. Cette méthode a montré son intérêt pour formaliser par exemple des comportements d'équipe dans le cadre du championnat de robots footballeurs.

1.7.2.2 Le modèle Aalaadin ou AGR

Le modèle Aalaadin de J. Ferber et O. Gutknecht [Gutknecht, 01], aussi appelé modèle AGR, considère le point de vue organisationnel. L'emploi de structures organisationnelles permet de garder une cohérence globale et facilite l'intégration et l'exécution simultanée d'agents hautement hétérogènes dans une même plateforme. A dessein, la sémantique d'un agent est minimaliste. Il est une entité autonome communicante qui joue des rôles au sein de différents groupes. Le groupe est un moyen d'identifier par regroupement un ensemble d'agent. Chaque agent peut être membre d'un ou plusieurs groupes. Un point majeur est que les différents groupes peuvent se recouper librement.
Voici ce que la définition formelle d'Aalaadin [Ferber & all, 00] a précisé :
La création d'un groupe est effectuée par un unique agent nommé le GroupServer et l'admission à un groupe est le fait d'agents gestionnaires de groupes responsables de l'évaluation des fonctions de rôles pour son groupe. Chaque gestionnaire de groupe est le membre maître des choix d'entrée d'autres agents.
Le rôle est une représentation abstraite d'une fonction, d'un service ou d'une identification d'un agent au sein d'un groupe particulier. Chaque agent peut avoir plusieurs rôles, un même rôle peut être tenu par plusieurs agents, et les rôles sont locaux aux groupes. L'hétérogénéité des situations d'interaction est rendue possible par le fait qu'un agent peut avoir plusieurs rôles distincts au sein de plusieurs groupes, les politiques de communications étant locale à un groupe.
Un agent agissant dans un groupe n'a pas le moyen de connaître l'activité d'un autre groupe auquel il n'appartient pas. Plus encore, à moins qu'un autre agent ne lui ait transmis une information spécifique, il n'en connaîtra même pas l'existence.

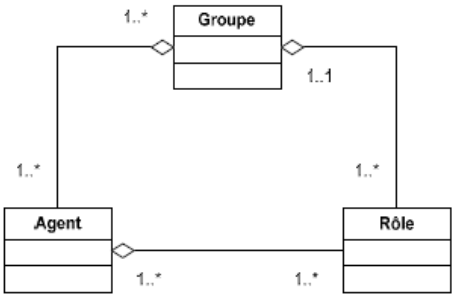


Figure 77 Le modèle AGR

Les comportements d'un agent sont structurés par les contextes des différents rôles que l'agent va tenir. Le comportement d'un agent est l'ensemble des comportements qu'il va associer à ses rôles.
Le comportement propre de l'agent assure la cohésion de tous les rôles joués par lui et définit la connaissance globale de l'agent, à savoir toute information ne se référant pas à un des rôles de l'agent.
Tous les rôles sont opaques les uns par rapport aux autres au sein d'un même agent, et tout partage d'information entre deux comportements de rôle transitent par le comportement propre.
Le groupe n'est pas un type d'agent particulier ayant une capacité de définitions récursives. De même, l'agent n'est pas un groupe particulier fédérant ses activités ou composants internes. De même, des idées de sous-groupes ne sont pas définissables directement dans le modèle puisque nous n'avons pas de fonctions liant directement un groupe à un autre. Ce n'est pas parce que le modèle n'accepte pas de notion intrinsèque d'agent récursif ou de relations entre groupes que de tels modèles y sont impossibles.

[Simonin & all, 02] L'étude et la conception de systèmes multi-robots (ou multi-agents situés) nécessite des outils de simulations performants et évolutifs appuyés sur les outils de conception de simulateur de la plate-forme MADKIT.

[Ferber & all, 05] L'extension AGRE intègre l'environnement en l'unifiant avec le modèle social. Un agent possède un certain nombre de représentants, appelés modes, qui sont ses « interfaces », ses manières d'être et d'agir dans un monde. Il existe deux types de modes : les modes physiques que l'on appelle « corps », et les modes sociaux, que l'on nomme « rôles » pour être cohérent avec AGR. Les rôles sont donc les manières sociales d'agir dans un groupe, alors que les corps sont les manières physiques d'agir dans une zone ou partie de l'environnement.

Le modèle AGR n'est pas une méthode mais contribue à l'analyse de l'organisation d'un SMA à spécifier.

1.7.2.3 La méthode GAIA

La méthode GAIA [Wooldridge & all, 00] fait reposer un système multi-agents sur les interactions entre les différents rôles présents dans le système. Elle est basée sur la constatation que les techniques classiques de génie logiciel, notamment les approches orientées objets, ne sont pas appropriées à une programmation orientée agent du point de vue de l'autonomie, des interactions et de la complexité de la structure organisationnelle des SMA. Le but de cette méthodologie est le même que celui d'AGR : capturer la structure organisationnelle du système. Elle permet de parcourir systématiquement le chemin qui commence par l'énoncé des demandes du problème et mène à une conception assez détaillée pour être implémentée tout de suite (le nom Gaia vient de l'hypothèse faite par James Lovelock qui dit qu'on peut voir tous les organismes de la biosphère comme participant ensemble à la régulation de l'environnement). Les auteurs proposent une méthodologie de conception de SMA basée sur une décomposition en deux niveaux : le niveau abstrait et le niveau concret.
Le niveau abstrait contient un modèle de rôles et un modèle d'interactions :

1. Le modèle de rôles décrit les différents rôles du système.
2. Le modèle d'interaction définit une liste de protocoles ; décrivant les communications possibles entre les rôles, ils sont définis par un initiateur, un interlocuteur, des entrées, des sorties ainsi qu'une description textuelle sur le type d'interaction et son déroulement.

La phase d'analyse consiste en un aller-et-retour entre ces deux modèles pour obtenir un ensemble cohérent. Elle est suivie d'une phase de conception dont le but est de ramener la description du système à un niveau d'abstraction suffisamment bas pour que les techniques traditionnelles de conception puissent être employées.

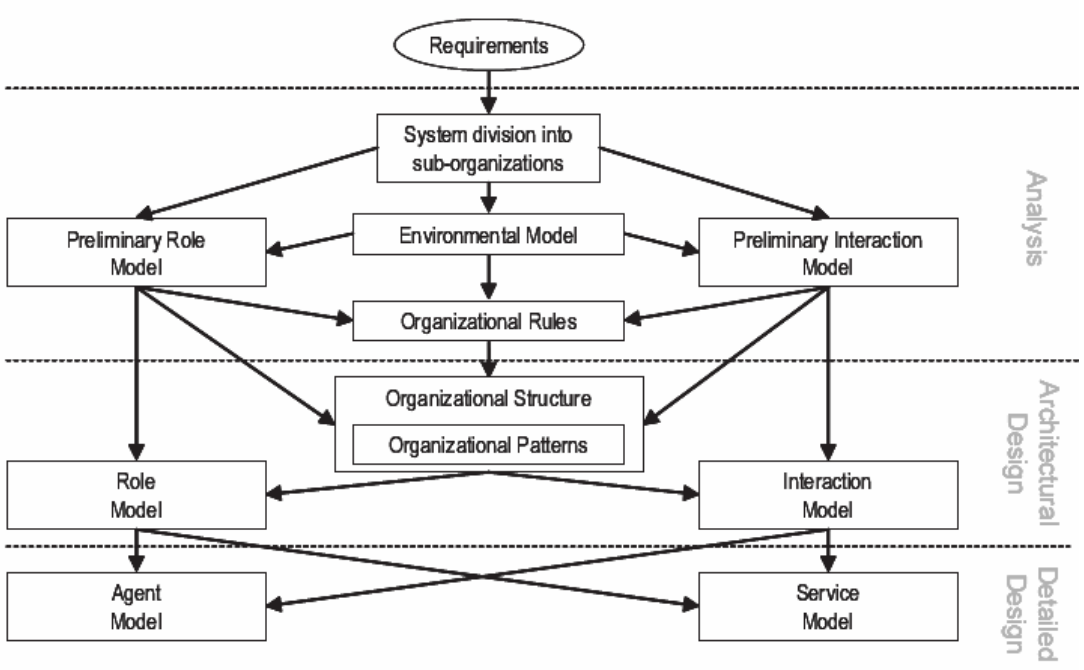


Figure 78 Les différentes étapes de la méthodologie Gaia

Un rôle est défini par quatre attributs :

- les responsabilités : ce sont les invariants du rôle répartis en *liveness* (propriétés actives : ce que l'agent doit faire pour arriver à remplir ces responsabilités) et en *safety* (propriétés de sûreté : les conditions que l'agent doit maintenir pendant l'exécution des activités)
- les permissions : les droits accordés à ce rôle pour accéder aux informations.
- les activités : qui correspondent aux comportement « privé » de l'agent.
- les protocoles d'interaction.

Afin de réaliser les responsabilités, un rôle a un ensemble de permissions qui sont des droits associés à un rôle. Les permissions d'un rôle identifient les ressources disponibles pour ce rôle pour réaliser les responsabilités. Les activités d'un rôle sont des calculs associés au rôle qui peuvent être fait par un agent sans interaction avec d'autres agents. Ce sont en quelque sorte des actions privées. Un rôle est aussi identifié par un nombre de protocoles qui définissent de quelle façon ce rôle va interagir avec d'autres rôles.

Le niveau concret contient les notions de type d'agent, de service et d'accointance et correspond à la conception.

1.7.2.4 La méthode Voyelles (A, E, I, O)

L'approche Voyelles [Ricordel, 01] consiste à considérer que l'analyse, le design, l'implémentation et le déploiement d'un système multi-agents peuvent être étudiés en fonction de quatre aspects fondamentaux : Agents, Environnement, Interaction et Organisation.
En voici une description simplifiée :
– Agents : architectures internes des agents.
– Environnement : le milieu dans lequel évoluent les agents.
– Interaction : les moyens par lesquels les agents interagissent.
– Organisation : les moyens utilisés pour structurer l'ensemble des entités.
L'idée sous-tendue par cette approche est que chacune de ces briques définit une problématique particulière qui doit être explicitement étudiée dans chaque étape de la conception, de la phase d'analyse à l'implémentation. La plate-forme de développement Volcano est basée sur cette méthodologie [Ricordel & all, 02]. Elle a par exemple été utilisée dans [da Silva & all, 02] où les auteurs appliquent la décomposition Voyelles pour l'étude de la coordination dans le contexte de la RoboCup.

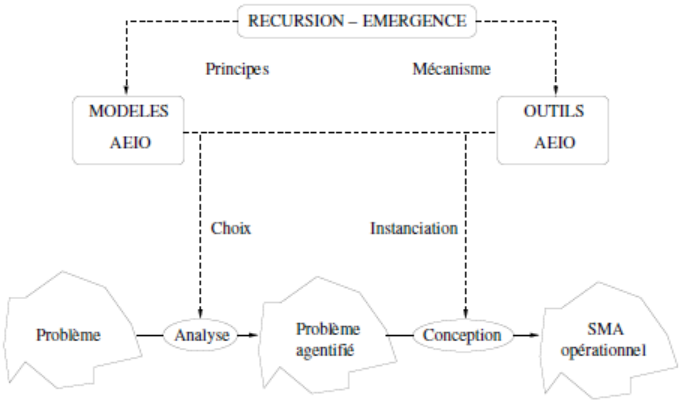


Figure 79 Les différentes étapes de la méthodologie Voyelles, tiré de [Seq, 03]

L'approche Voyelles, dans le principe déclaratif, compose donc ces quatre briques pour former un système multi-agents.
SMA = Agents + Environnement + Interactions + Organisations
Dans le principe fonctionnel, les fonctionnalités d'un système multi-agents incluent les fonctionnalités individuelles des agents enrichies des fonctionnalités qui résultent de la valeur ajoutée par le système multi-agents lui-même, parfois appelée intelligence collective.
Fonction(SMA) = Fonctions(Agents) + Fonction collective
Le principe de récursion de cette approche vise à considérer un système multi-agents comme des entités multi-agents, abstraction manipulable par la programmation orientée multiagents.
SMA* = SMA
D'après le paradigme Voyelles, pour un problème à résoudre (ou un système à simuler) dans un domaine donné, l'utilisateur choisit le modèle d'agent, le modèle d'environnement, le modèle d'interaction et le modèle d'organisation qu'il instancie ensuite, ainsi que leurs dynamiques selon les trois principes Voyelles, de manière à engendrer le système multi-agents computationnel et déployable qui résoudra le problème dans le domaine considéré.
Cependant la prise en charge dynamique des rôles et des tâches est difficile à représenter. Les derniers développements de l'équipe MAGMA se focalisent sur l'intégration de l'Utilisateur dans Voyelles (A, E, I, O, U).

Pour conclure cette section, on peut remarquer que si certains modèles proposent la notion de rôles et d'agents, tous ne permettent pas à un agent de prendre en charge dynamiquement un rôle. La structure organisationnelle du système est souvent statique, les relations entre agents ne changent pas pendant l'exécution.

1.7.3 Vers une représentation unifiée des agents par UML

Pour terminer ce tour d'horizon de la programmation robot, agent, acteurs et parallèle, il convient de présenter les travaux récent sur UML dont l'objectif est d'uniformiser le langage des méthodes orientées multi-agents afin de faciliter le travail des outils de spécification.

1.7.3.1 UML 1

UML [Odell, 1998] (Unified Modeling Language) unifie et formalise les méthodes de plusieurs approches orientées objets, incluant Booch, Rumbaugh (OMT), Jacobson, et Odell.

UML prend en charge plusieurs types de modèles :
• les cas d'utilisation : la spécification des actions que le système ou la classe peut réaliser en interaction avec les acteurs extérieurs. Ils sont souvent utilisés pour décrire comment un utilisateur communique avec son logiciel ;
• les modèles statiques : ils décrivent la sémantique statique des données et des messages d'une manière conceptuelle et d'un manière plus proche de l'implémentation (il s'agit des diagrammes de classes et de packages) ;
• les modèles dynamiques : ils incluent les diagrammes d'interaction (i.e., les diagrammes de séquence et les diagramme de collaboration), les schémas d'état et les diagrammes d'activité ;

• les modèles d'implémentation : ils décrivent la répartition des composants sur différentes plateformes (i.e., les modèles de composants et les diagrammes de déploiement) ;
• le langage de contrainte par objet (OCL), qui est un langage formel simple pour exprimer la sémantique dans une spécification UML. Il peut être utilisé pour définir des contraintes sur le modèle, des invariants, des pré-conditions et des post-conditions d'opérations et des chemins de navigation dans un réseau de navigation.

Outre ces diagrammes classiques pour l'approche objet, on peut étudier les interactions entre objets à l'aide des diagrammes de séquences et de collaboration.

Les diagrammes de séquences
Le diagramme de séquences modélise les interactions. Il s'utilise essentiellement pour décrire les scénarios d'un cas d'utilisation (les entités sont les acteurs et le système) ou décrire des échanges de messages entre objets.

Les diagrammes de collaboration
Le diagramme de collaboration est une mise en contexte des interactions des entités du système : on peut y préciser les états des entités qui interagissent. Dans le chapitre 2, nous allons utiliser ce type de diagramme. Aussi nous proposons quelques explications supplémentaires utiles à la bonne compréhension de ces diagrammes tels que définis dans [Gomaa, 00], ouvrage UML 1.x consacré aux applications concurrentes, distribuées et temps-réels.

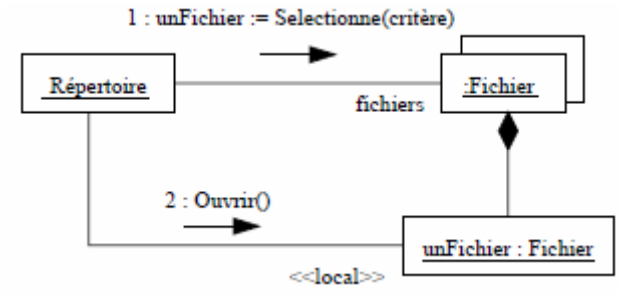


Figure 80 Diagrammes de collaboration dans [Gomaa, 00]

Sur ce diagramme, on peut indiquer pour chaque message les clauses qui conditionnent son envoi, son rang (son numéro d'ordre par rapport aux autres messages), sa récurrence et ses arguments.
La syntaxe d'un message est la suivante est `[pré "/"] [["["cond"]"] [séq] ["*"["||"]["["iter"]"]]":"] [r ":="] msg"("[par")"`
– `pré` sont les prédécesseurs (liste de numéros de séquence de messages séparés par une virgule).
Un message n'est envoyé que lorsque tous ses prédécesseurs le sont aussi (permet de synchroniser l'envoi de messages).
– `cond` est une garde, une expression booléenne, qui permet de conditionner l'envoi du message, à l'aide d'une clause exprimée en langage naturel.
– `séq` est un numéro de séquence du message qui indique le rang du message, c'est-à-dire son numéro d'ordre par rapport aux autres messages.
– `iter` qui indique une récurrence du message et permet de spécifier en langage naturel l'envoi séquentiel (ou en parallèle, avec "`||`") de messages.
– `r` qui est la valeur de retour du message,
– `msg` le nom du message.
– `par` les paramètres (optionnels) du message.

1.7.3.2 A-UML

Les faiblesses d'UML pour la représentation des systèmes multi-agents ont conduit une équipe de chercheurs travaillant dans différentes entreprises ou universités à concevoir A-UML.
L'objectif est de mettre au point des sémantiques communes, des méta-modèles et une syntaxe générique pour les méthodologies agents. A-UML est un des fruits de la coopération entre FIPA (Foundation of Intelligent Physical Agents) et l'OMG (Object Management Group).
A-UML n'est pas un langage de modélisation fini et adopté par la communauté car aucune spécification finale n'a été publiée officiellement mais plusieurs publications sur des extensions d'UML ont été publiées par les membres du groupe. Proposé notamment par Odell [Odell & all, 05], A-UML complète UML par des extensions pratiques et spécifiques aux SMA en supprimant les aspects trop spécifiquement objets.

1.7.3.3 Nouveau standard UML 2.1

AUML avant UML 2.0 définissait des rôles, des points de décision, la concurrence, la modularité, la communication simultanée avec un groupe d'agents (multicast) comme pour l'exemple de la Figure 81a. UML 2.0 inclut toutes ces notions à l'exception des rôles et du multicast. Sections critiques, boucles, alternatives, parallélisme et séquence ont été intégrées.
Les diagrammes d'interactions d'UML (diagrammes de séquences, diagrammes de collaboration/communication) contiennent la même information et ne diffèrent que par leur représentation graphique : les premiers portent l'importance sur la chronologie de la séquence de communication, tandis que les seconds la portent sur les associations parmi les agents.
Dans les diagrammes de séquence, un symbole en forme de diamant est ajouté à UML afin de représenter l'éventail des choix d'interactions possibles pour un agent à un moment précis de l'interaction.
Pour la communication simultanée et les multi réponses, une notation basée sur des cardinalités est ajoutée aux lignes de messages. Par exemple, dans la Figure 81b, le message `cfp` est annoté pour indiquer un message, qui peut être envoyé simultanément à n participant. La réponse consiste au refus de j participants et en une proposition de k autres participants. Des contraintes entre les différentes cardinalités ne sont pas encore possibles dans UML 2.0.

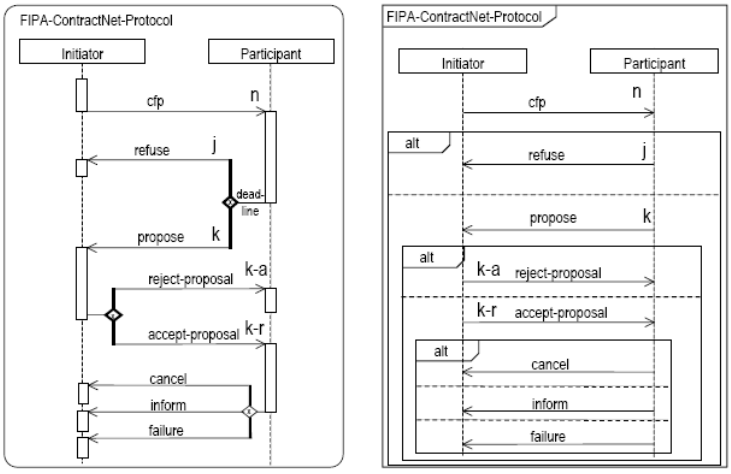


Figure 81 Les diagrammes de séquence dans (a) UML 1 étendu puis dans (b) UML 2.0

Les diagrammes de communication (autrefois nommés diagrammes de collaboration dans UML 1.0) s'opposent aux diagrammes de séquence dans ses objectifs : ils représentent l'organisation du dialogue entre agents et placent à un niveau secondaire la mise en valeur de la chronologie et du parallélisme. Ainsi l'historique du dialogue est peu aisé mais interactions directes ou indirectes entre agents sont plus visibles.

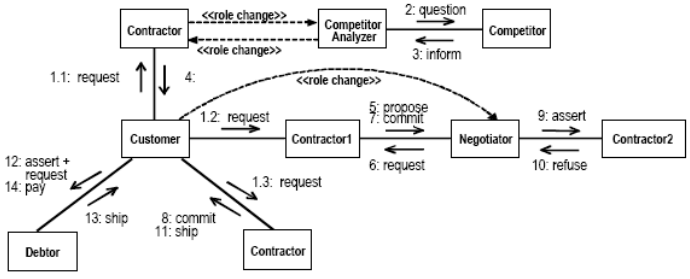


Figure 82 Diagramme de communication dans UML 2.0

Par rapport à A-UML, les notions de groupes de fait pas partie d'UML 2.0. Il reste des limites au niveau des notions de rôles des agents. Surtout pour le changement de rôles qui sont mal pris en compte.

Les diagrammes d'activité sont centrés sur la séquence et les conditions de coordinations des comportements de bas niveau. De fait un plan peut être représenté par un diagramme d'activité. Cependant une règle de plan devrait être précisée pour exprimer les conditions de déclenchement. Elle n'est pas encore prévue selon Odell dans UML 2.0.
La notion de rôle peut être exprimée dans un diagramme d'activité selon 2 manières : par partition et à l'aide d'une notation. Il en est de même pour la notion de groupe.

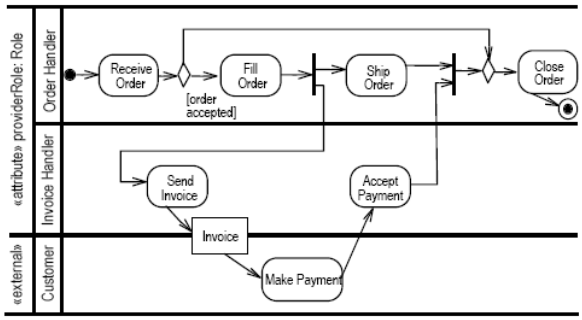


Figure 83 Un diagramme d'activité partitionné

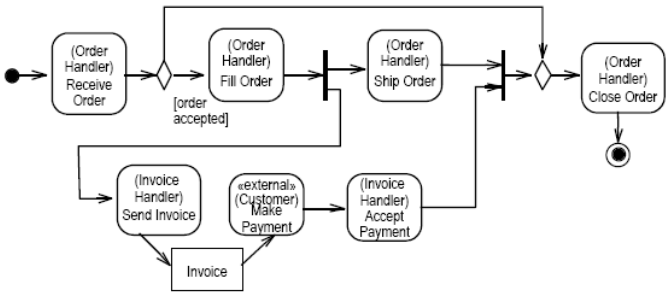


Figure 84 Un diagramme d'activité avec les rôles annotés

Les équipes impliquées dans AUML étudient le nouveau standard UML 2.1, s'appuient aussi sur le langage de l'OMG SysML (Systems Modeling Language) qui permet l'adaptation d'UML, et sur UPMS (UML Metamodel and Profile for Services) pour faire de nouvelles propositions. Le standard en voie de définition UMPS contient une proposition pour inclure dans UML les concepts agents.

En conclusion de cette partie, c'est qu'il existe des réflexions certainement utiles à une bonne programmation MASL. Et les modèles et les méthodes exposées se plaçant délibérément dans la programmation multi agents ne peuvent qu'être la source principale pour définir une méthodologie pour le programmeur MASL pour aboutir à une programmation plus sure.

1.8 Synthèse

Notre objectif est de programmer un ensemble hétérogène de robots en respectant six propriétés, notamment le langage doit :

1. permettre de prendre en compte l'hétérogénéité des agents ;
2. posséder des types d'exécution pour rythmer l'activité d'un groupe au sein d'une équipe ;
3. proposer différents modes de communication entre robots ;
4. introduire des « entrées dans le code » qui rendent simple les changements d'équipes ;
5. offrir un dispositif dit de « perméabilité » qui garantit l'autonomie des agents et la mise en œuvre de différents modes d'exécution et notamment la définition de modes d'exécution dégradés ;
6. permettre l'extensibilité : un programme pour un groupe de robot doit pouvoir s'exécuter pour un plus grand nombre de robots sans perte brutale de performance.

On peut dresser un résumé de la couverture de ces propriétés dans les travaux précédents concernant les langages d'agents.

Tableau 2 Langages orientés agents robotiques et les six propriétés

	(1)	(2)	(3)	(4)	(5)	(6)
CHARON	+	Pas aussi précis	SV, MP, E-	Pas aussi précis	+	-
CCL	+	Pas aussi précis	SV, MP	Pas aussi précis	+	+
MRL	+	Pas aussi précis	E, SV, MP		+	-
Tapir	+	Pas aussi précis	MP, SV		+	-
GOLOG	+	Pas aussi précis	E, SV		+	-
CDL	+	Pas aussi précis	E, MP	-	-	+
XABSL	+	Pas aussi précis	E, MP	Pas aussi précis	-	+
HoRoCoL	-	Manque simplement le mode scalaire. Pas d'imbrication de blocs parofseq/ seqofpar	E,SV,MP	+	-	+

Légende: SV (Shared Variable), MP (Message Passing), E (Events), E- (partially implemented), + (feature available), - (feature unavailable), nothing (not documented).

De ce que l'on a pu constater dans ce chapitre, on retiendra :

- des langages data-parallèles les méthodes de synchronisme d'exécution qui serviront a l'implémentation de MASL ;
- des langages acteurs les messages asynchrones et l'interface dynamique qui introduiront la notion de perméabilité en MASL ;
- des langages réactifs le synchronisme et l'asynchronisme de réaction au changement de l'environnement et le broadcast d'événements que MASL couvrira avec les « `events` » ;
- des architectures d'agents robotique doivent pouvoir être implémentée ;
- des systèmes multi-agents la coopération, l'interaction et la communication que MASL couvrira avec les variables partagées ou les `events` ;
- de l'autoreconfiguration robotique la nécessité de prendre en compte dynamiquement les changements de rôles ou de groupe que MASL couvrira avec les « entry ».

Notre modèle de programmation est une unification de plusieurs approches concernant les modèles d'agents (délibératifs, réactifs, hybrides), les modèles de communication (par mémoire partagée, par évènements, par envoi de messages synchrones, par envoi de messages asynchrones), les modèles de contrôle d'exécution et de synchronisation (séquence, exécution parallèle asynchrone, exécution parallèle synchrone) dans le contexte multi-robot. Pour une vision offerte au programmeur simplifiée, une seule construction « `entry` » qui peut être imbriquée, permet de mixer les différentes approches. La portée des variables, la destination des évènements sont bornées par la syntaxe du bloc `entry` et sont définis dynamiquement à l'exécution : les agents qui s'exécutent actuellement dans le bloc où ces derniers ont été déclarés, ont accès aux variables et sont destinataires des évènements. De même l'accès aux membres des agents est conditionné à son état de perméabilité courant et aux rôles respectifs courants joués par l'agent demandeur par rapport à l'agent considéré. S'ils s'exécutent actuellement dans un même bloc entry, l'agent appliquera la politique de collaboration par rapport à un « collègue » plutôt que celle définie pour un autre agent quelconque. De même s'ils s'exécutent dans un même bloc `entry` synchrone, ils partageront une même barrière de synchronisation.
Les agents embarquent le code MASL traduit dans leur langage avec la possibilité d'appeler un runtime MASL. Les agents qu'ils soient actifs ou réactifs sont contrôlés via leur A.P.I. On « agentifie » des robots existants en leurs fournissant un programme de contrôle qui s'appuie sur le concept de perméabilité pour que soit respectée une autonomie relative selon l'acception MAS.

Chapitre 2 : Modèle de programmation unifié pour le contrôle d'agents robotiques : présentation par l'exemple du langage MASL

Notre point de vue repose sur une abstraction des modèles d'agents, des modèles de communication, des modèles de synchronisation issus des travaux cités dans le chapitre 1. Parmi les modèles d'agents, on distingue les agents purement délibératifs, purement réactifs et les agents hybrides. Flots de données, data parallélisme, parallélisme de contrôle sont des modèles de concurrences. Communication point-à-point, par mémoire partagée, par diffusion d'évènements sont les modèles de communications considérés. Unifier ces modèles conduit à des systèmes plus ouverts et plus facilement réutilisables. L'utilisation de XML pour l'intégration des API existantes contribue aussi à l'ouverture des systèmes.

L'objectif de ce chapitre est d'introduire par l'exemple les concepts clefs et originaux de notre contribution :

2.1 les fichiers XML qui permettent l'hétérogénéité des agents et l'indépendance par rapport à leur runtime ;

2.2 l'intégration du parallélisme synchrone et asynchrone ;

2.3 la communication par variables partagées et par évènements

2.4 le renforcement de l'équipe par rattachement dynamique d'un agent à une mission en cours;

2.5 la perméabilité qui définit dynamiquement le degré de protection et de visibilité de l'agent i.e. sa capacité à adapter son interface.

2.1 MASL et l'hétérogénéité

Afin de pouvoir travailler dans un contexte multi-robots qui peuvent être potentiellement différents, nous considérons une abstraction d'API (Application Programming Interface) par classe d'agent qui permet de décrire dans un modèle unifié des agents conçus sur des modèles d'exécution, de communication et de synchronisation variés.

Cette API est décrite par un fichier XML qui permet de définir :

- les méthodes (procédures et fonctions) ;
- les attributs qui permettent d'accéder à l'état de l'agent ;
- la perméabilité qui définit la politique de coopération de l'agent en fonction de son contexte.

Nous allons illustrer l'utilisation des services offerts par l'API des agents. Lorsque les agents sont de types différents mais partagent des membres, le polymorphisme contribue à unifier la programmation de la mission.

2.1.1 API et fichiers XML

Supposons que nous ayons à programmer 2 types de robots : un robot à pattes `MAAM` qui sait se déplacer en avant, en arrière, tourner à gauche, tourner à droite, et un autre agent, un robot à roues `Khepera` qui en plus possède des capteurs de proximités et des informations sur le chargement de sa batterie.

On dispose donc des interfaces suivantes illustrées dans le diagramme de classes UML suivant. L'extrait ne concerne que les premiers exemples de ce chapitre. Pour le premier exemple, seules les classes `Robot`, `MAAM` et `Khepera` sont à considérer.

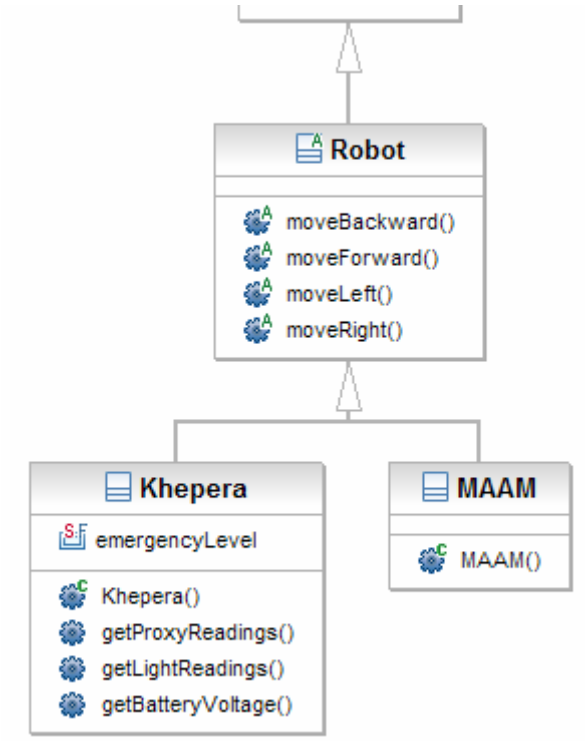


Figure 85 Diagramme de classes du chapitre 2

Ces descriptions sont enregistrées dans des fichiers XML `Khepera.xml` & `Maam.xml`. Cf. Annexe

2.1.2 Instanciations d'agents hétérogènes

Rôle du programme MASL :

Instancier pour la mission des agents de types différents.

Pseudo code de cet exemple :

```
Instancier 2 agents de type Khepera et 3 agents de type MAAM
```

Code MASL correspondant :

```
01| import Khepera.xml as Khepera;
02| import MAAM.xml as MAAM;

03| Khepera k1,k2 = newAgent(Khepera);
04| MAAM m1, m2, m3 = newAgent(MAAM);
```

Explication du code :

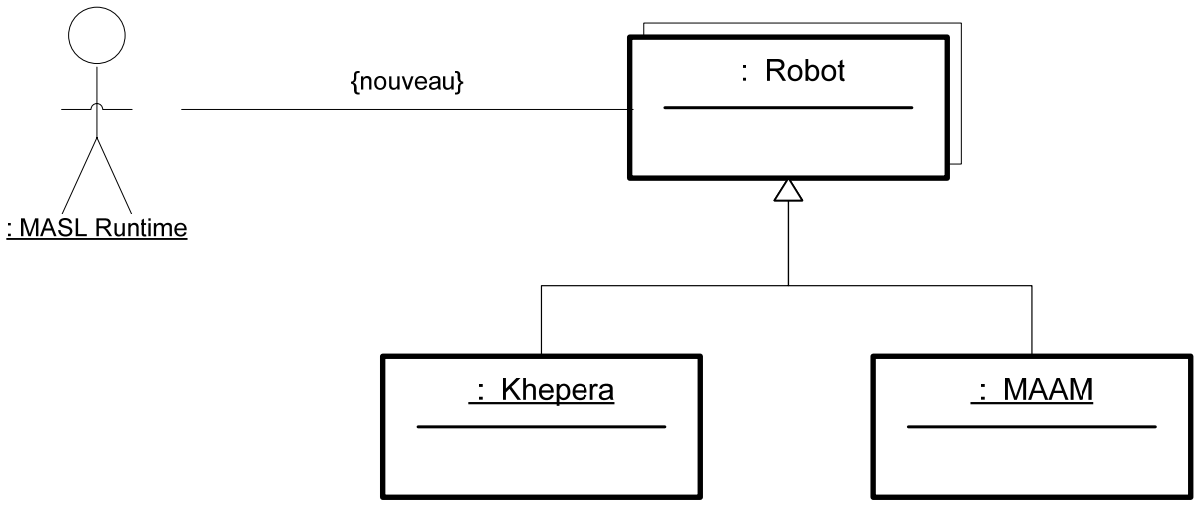
Ligne 1 : Définition d'un `type Khepera` qui importe son interface du fichier `Khepera.xml`

Ligne 2 : Définition d'un `type MAAM` qui importe son interface du fichier `MAAM.xml`

Ligne 3 : Déclaration de deux variables `k1` et `k2` de `type Khepera` qui sont instanciées à l'aide du constructeur définit dans le fichier XML.

Ligne 4 : Déclaration de trois variables `m1, m2` et `m3` de type `MAAM` qui sont instanciées à l'aide du constructeur définit dans le fichier XML.

Le diagramme de collaboration :



L'acteur `MASL Runtime` instancie une collection d'agents considérés comme des objets actifs. `Robot` étant une classe abstraite, il n'y a pas d'instances. Seules existent les instances de `Khepera` et de `MAAM`. La notation est tirée de [Gomaa, 00], *Cf.* chapitre 1.

Vision offerte au programmeur :

```
01| Robot[] a=new Robot[main.count];
02| for (i=0;i<2;i++)
03|   a[i]=new Khepera();
04| for (i=2;i<5;i++)
05|   a[i]=new MAAM();
06| Khepera k1=a[0];
07| Khepera k2=a[1];
08| MAAM m1=a[2];
09| MAAM m2=a[3];
10| MAAM m3=a[4];
```

Comme nous le verrons plus loin, le tableau implicite `a[]` *sera utilisé notamment pour la traduction en séquentiel des programmes parallèles. Le langage MASL permet alors de manipuler de manière naturelle et implicite des collections. La variable* `main.count` *donne le cardinal de l'ensemble des agents présents dans la mission. Dans la mesure où MASL permet de mixer références implicites et références explicites, les références explicites* `k1`, `k2`, `m1`, `m2` *et* `m3` *sont aussi déclarées.*

Le tableau `a[]` *permet d'utiliser le polymorphisme implicite dans MASL car il est déclaré à l'aide de la classe ancêtre des types* `MAAM` *et* `Khepera`*. Les agents sont concrètement instanciés dans ces types. Ainsi par le mécanisme de look-up dynamique, la redéfinition des méthodes communes ou l'implémentation des méthodes abstraites seront pris en compte de la manière la plus fine au niveau de chaque instance.*

2.2 MASL et le parallélisme

Nous présentons dans cette partie comment MASL prend en compte les modèles de parallélisme.

Dans ce paragraphe nous présentons l'instruction **entry** qui défini des portes dans le code par lesquelles les agents peuvent entrer. Comme nous allons l'illustrer, dans les exemples qui suivent, une entry est caractérisée par 3 éléments :

- un **identifiant** qui permet de nommer cette porte ;
- son **modèle de parallélisme** : `scalar` (modèle séquentiel), `synchronous` ou `asynchronous` **;**
- un **test** qui permet aux agents de vérifier par eux-mêmes s'ils doivent ou non entrer par cette porte.

2.2.1 Le parallélisme asynchrone

Avec les deux fichiers XML précédents, nous allons pouvoir présenter un 1er exemple dans lequel nous illustrons la mise en parallèle de 5 agents.

Rôle de l'algorithme :

Instancier pour la mission des agents de types différents puis exécuter sur chacun d'eux une séquence identique d'appels de méthodes de leur API.

Principe de l'algorithme :

Utiliser l'anonymat (références implicite) pour faire abstraction des spécificités propres à chaque type. L'exécution parallèle d'un même programme sur 5 agents différents est indépendante des autres.

Pseudo code de cet exemple :

```
Instancier 2 agents de type Khepera et 3 agents de type MAAM.
Exécuter sur tous les agents une séquence de 4 appels à leur API.
```

Code MASL correspondant :

```
01|
…  //voir exemple précédant
04|

05| asynchronous entry main (true) {
06|    .moveLeft(30);
07|    .moveForward(10);
08|    .moveRight(30);
09|    .moveBackward(10);
10| }
```

Explication du code

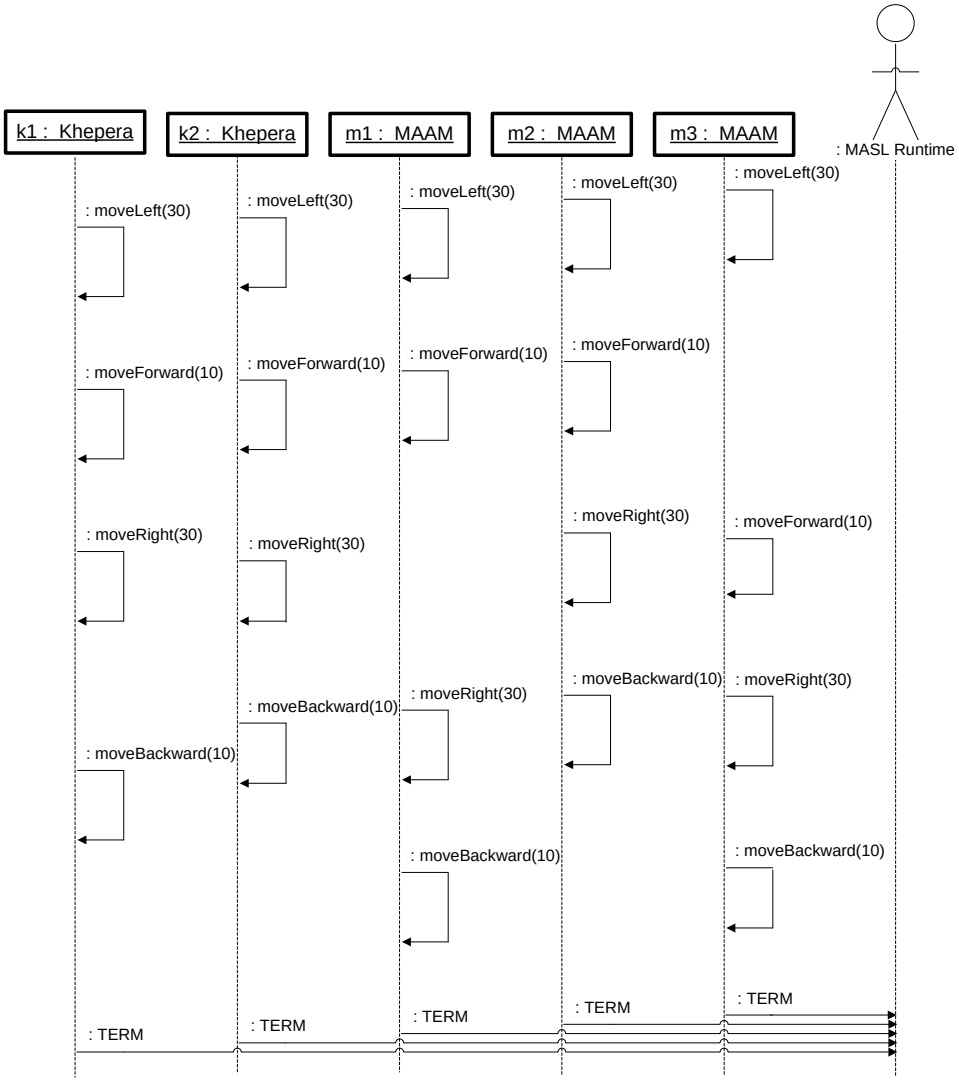
Ligne 5 : La programmation de la mission des agents se fait par une instruction **asynchronous entry main (true)**. Le terme `asynchronous` permet de préciser que chaque agent exécute son code de manière parallèle indépendamment des autres. Tous les entrelacements sont possibles. L'expression `true` signifie que le programme concerne inconditionnellement chaque agent instancié précédemment. Le terme `main` est le nom de la porte et désigne qu'il s'agit du point d'entrée du programme de chaque agents.

Lignes 6-9 : Les agents `k1`, `k2` de type `Khepera` et `m1`, `m2`, `m3` de type `MAAM` exécutent leur code en parallèle.

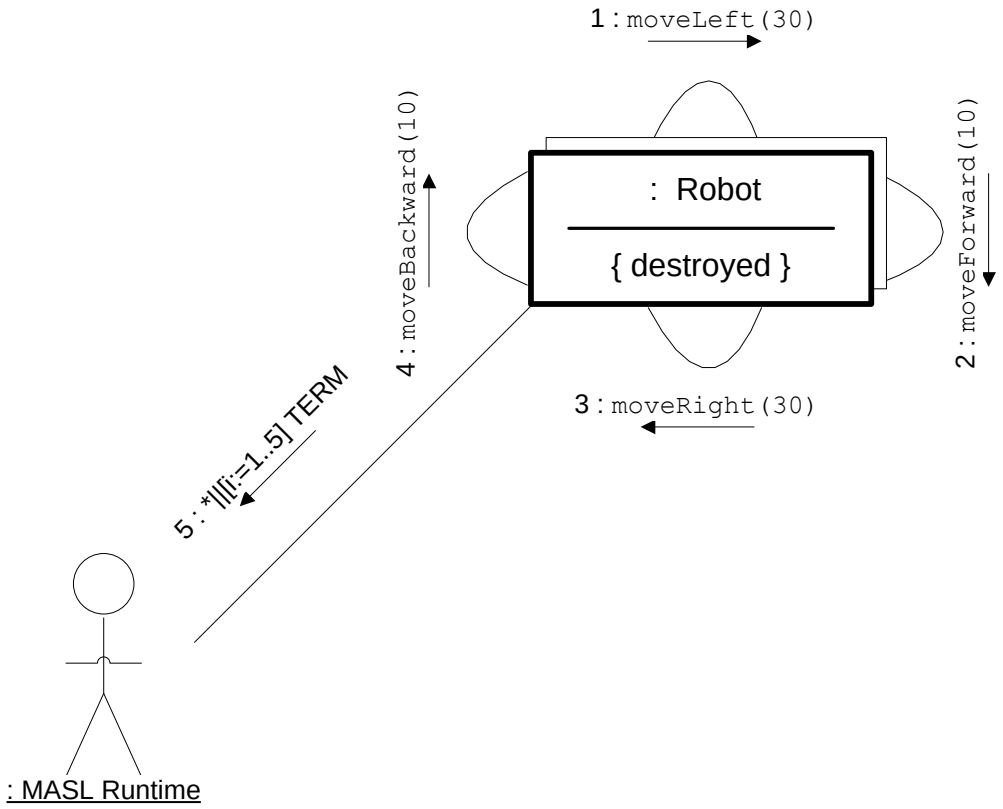
En cas d'appel sur une méthode `moveLeft()`, un polymorphisme est appliqué.

Le diagramme de séquence :

Ce graphique ne fait aucune hypothèse sur l'ordre exact d'exécution des différentes primitives. Il ne présente qu'un ordonnancement possible.



Le diagramme de collaboration :



Une autre présentation définie par le langage UML qui se focalise sur les classes plus que sur les instances.
A noter que diagramme met l'accent sur l'absence de collaboration dans l'exemple.

Vision offerte au programmeur :

```
01|
      Tous les agents considérés sont rangés dans un tableau d'agents
      Robot (ancêtre des deux classes d'agents) de taille
      correspondant à la valeur de main.count référencé par la variable a.
04|

05| for (i=0 ; i< main.count; i++) {
06|   if (true) {                          // asynchronous entry main
07|     a[i].moveLeft(30);
08|     a[i].moveForward(10);
09|     a[i].moveRight(30);
10|     a[i].moveBackward(10);
11|   }
12| }
```

Il s'agit d'un ordonnancement particulier où les agents exécutent la totalité de leur programme en série. Cette absence de concurrence n'est pas optimale. Mais cet ordonnancement permet l'exécution correcte de l'algorithme.

Problèmes liés :

Le polymorphisme (*Cf. Chapitre 3*) avec son mécanisme de look-up dynamique qui choisi sur chaque agent quelle implémentation est invoquée.

2.2.2 Le parallélisme synchrone

Sur cet exemple, nous allons illustrer le concept de parallélisme synchrone qui correspond à une exécution coordonnée des instructions par les agents du bloc.

Rôle de l'algorithme :

Instancier pour la mission des agents de types différents puis exécuter sur chacun d'eux une séquence identique d'appels de méthodes de leur API. Chaque agent attend les autres pour passer en séquence.

Principe de l'algorithme :

Utiliser l'anonymat (références implicite) pour faire abstraction des spécificités propres à chaque type. L'exécution parallèle d'un même programme sur 5 agents différents est dépendante de chaque agent : A la terminaison d'une méthode appelée, l'agent attend que les autres aient terminé leur appel.

Pseudo code de cet exemple :

```
Instancier 2 agents de type Khepera et 3 agents de type MAAM.
Exécuter sur tous les agents une séquence synchronisée de 4 appels à leur
API
```

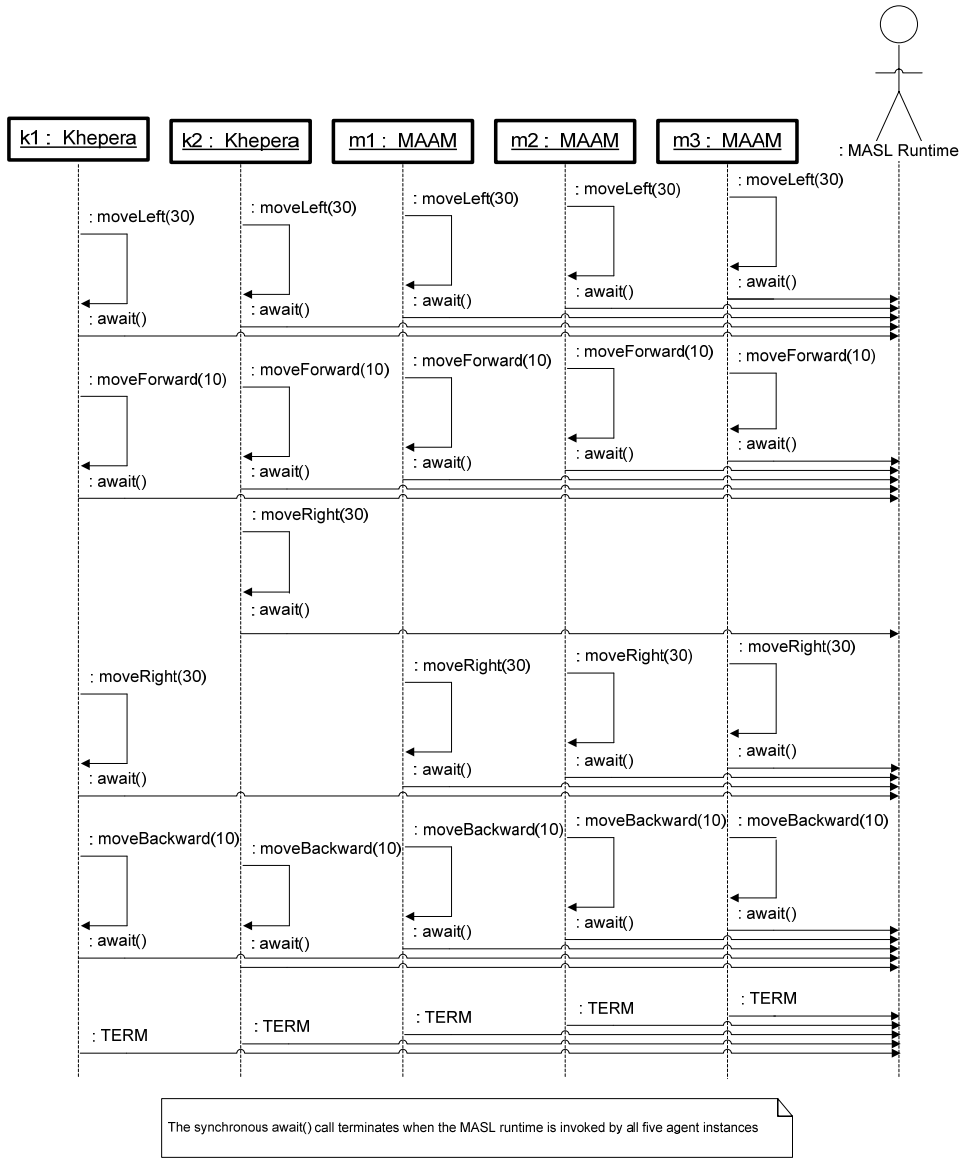
Code MASL correspondant :

```
01| import Khepera.xml as Khepera;
02| import MAAM.xml as MAAM;

03| Khepera k1,k2 = newAgent(Khepera);
04| MAAM m1,m2,m3 = newAgent(MAAM);

05| synchronous entry main (true) {
06|    .moveLeft(30);
07|    .moveForward(10);
08|    .moveRight(30);
09|    .moveBackward(10);
10| }
```

Explication du code :

Ligne 5 : les 5 agents `k1`, `k2`, `m1`, `m2`, `m3` entrent par la porte `main`.
Ligne 6 : les 5 agents `k1`, `k2`, `m1`, `m2`, `m3` exécutent `moveLeft(30)` puis s'attendent mutuellement.
Ligne 7 : les 5 agents `k1`, `k2`, `m1`, `m2`, `m3` exécutent `moveForward(10)` puis s'attendent mutuellement.
Ligne 8 : les 5 agents `k1`, `k2`, `m1`, `m2`, `m3` exécutent `moveRight(30)` puis s'attendent mutuellement.
Ligne 9 : les 5 agents `k1`, `k2`, `m1`, `m2`, `m3` exécutent `moveBackward(10)` puis s'attendent mutuellement.

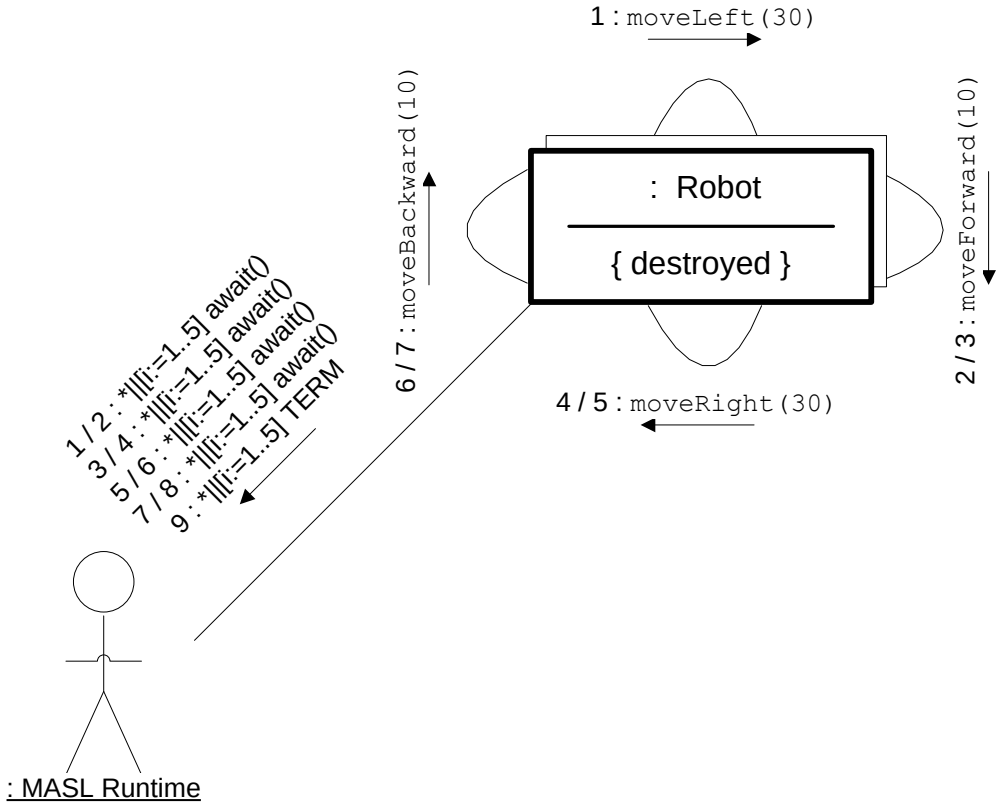
Le diagramme de séquence :



Le diagramme de séquence illustre un des moyens de réaliser l'attente mutuelle des agents : L'exécutif MASL est invoqué par une méthode bloquante `await()` qui ne termine que si tous les agents l'ont invoqué.

L'exécution synchrone garantie que chaque agent attend la terminaison de l'exécution de l'instruction des autres pour passer en séquence. Par contre, elle ne garantit pas que le démarrage ou les terminaisons de chacune de ces instructions aient lieu au même moment.

Le diagramme de collaboration :



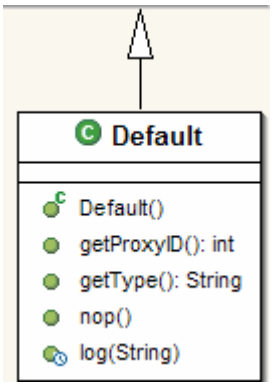
Vision offerte au programmeur :

```
01|
      Tous les agents considérés sont rangés dans un tableau d'agent
      Robot (ancêtre des deux classes d'agents) référencé par la variable
      a.
04|

05| for (i=0 ; i< main.count; i++)
06|   if (true)                          // synchronous entry main
07|     a[i].moveLeft(30);
08| for (i=0 ; i< main.count; i++)
09|   if (true)                          // synchronous entry main
10|     a[i].moveForward(10);
11| for (i=0 ; i< main.count; i++)
12|   if (true)                          // synchronous entry main
13|     a[i].moveRight(30);
14| for (i=0 ; i< main.count; i++)
15|   if (true)                          // synchronous entry main
16|     a[i].moveBackward(10);
17|
```

Il s'agit d'un ordonnancement particulier où tous les agents exécutent leur programme lignes par lignes. Cette absence de concurrence n'est pas optimale. Mais cet ordonnancement permet l'exécution correcte de l'algorithme.

2.2.3 Une entrée scalaire

Cet exemple sert à montrer le sens de l'élément `scalar` d'une entrée



Rôle de l'algorithme :

Afficher « Hello World ! » une seule fois, indépendamment du nombre d'agents.

Principe de l'algorithme :

Afficher d'abord « Hello » puis afficher « World ! ».

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2 = newAgent(Default);

03| scalar entry main (true) {
04|    .log("Hello ");
05|    .log("World !\n");
06| }
```

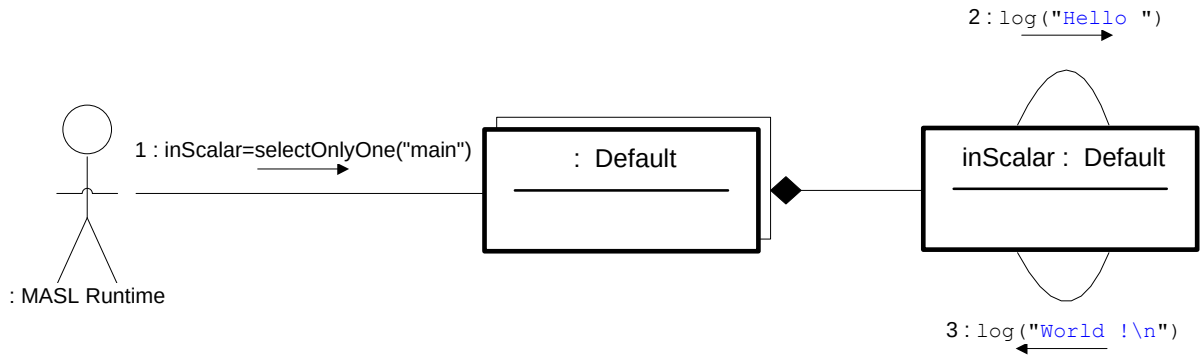
Explication du code :

Ligne 1 : Définition d'un `type Default` qui importe son interface du fichier `Default.xml`
Ligne 2 : Déclaration de deux `variables a1` et `a2` de `type Default` qui sont instanciées à l'aide du constructeur défini dans le fichier XML.
Ligne 3 : parmi tous les agents instanciés, un seul agent entre dans le bloc, c'est le sens du mot `scalar`.
Ligne 4 & 5: On a une seule exécution de la séquence.

Affichage du programme dans un seul fichier de journalisation :

Seule possibilité
`Hello World !`

Le diagramme de collaboration :



Une instance particulière de la collection d'agent `Default`, celle qui se trouve dans le bloc `scalar`, exécute sa méthode `log()` en séquence.

Vision offerte au programmeur :

```
01|
      Tous les agents considérés sont rangés dans un tableau d'agents
      Default (ancêtre de toutes les classes d'agents) référencé par
      la variable a.
03|
04|   if (true) {                          // scalar entry main
05|     this.log("Hello ");
06|     this.log("World !\n");
07|   }
```

2.3 MASL et la communication

2.3.1 La communication par variables partagées

Rôle de l'algorithme :

Manipulations de variables partagées MASL. Illustrer la différence avec les variables MASL locales.

Principe de l'algorithme :

Accès en lecture puis en écriture de variables MASL partagées ou locales.

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3 = newAgent(Default);

03| asynchronous entry main (true) {
04|   shared int svar=0;
05|   local int lvar=0;
06|   lvar++;
07|   svar++;
08|   .log("lvar="+lvar+"\n");
09|   .log("svar="+svar+"\n");
10| }
```

Explication du code :

Ligne 4 : La variable `svar` (préfixée par « `s` » pour `shared`) est partagée par tous agents qui donc accèdent qu'à un unique exemplaire de cette variable. Des problèmes d'accès concurrents sont alors possibles. Dans le cas présent, la portée de cette variable est toute la mission. On pourrait rapprocher une variable partagée à une variable de classe prévue dans l'API d'un type d'agent.

Ligne 5 : La variable `lvar`, préfixée par « `l` » pour `local`, est disponible pour chaque agent. Il est impossible pour un agent d'accéder à une variable locale d'un autre agent. On pourrait rapprocher une variable locale à une variable d'instance prévue dans l'API d'un type d'agent.

Lignes 6 et 7 : Les variables `lvar` et `svar` sont incrémentées de 1 par les 3 agents.

Lignes 8 et 9 : Les 3 agents a1, a2 et a3 affichent les valeurs des deux variables.

Affichages possibles du programme dans un seul fichier de journalisation, suivant l'ordonnancement entre les agents a1, a2 et a3 :

Première possibilité	Deuxième possibilité	Troisième possibilité	Quatrième possibilité
lvar= 1	lvar= 1	lvar= 1	lvar= 1
svar= 1	lvar= 1	lvar= 1	svar= 1
lvar= 1	lvar= 1	svar= 1	lvar= 1
svar= 2	svar= 1	svar= 2	lvar= 1
lvar= 1	svar= 2	lvar= 1	svar= 2
svar= 3	svar= 3	svar= 3	svar= 3

Cinquième possibilité
lvar= 1 lvar= 1 svar= 1 lvar= 1 svar= 2 svar= 3

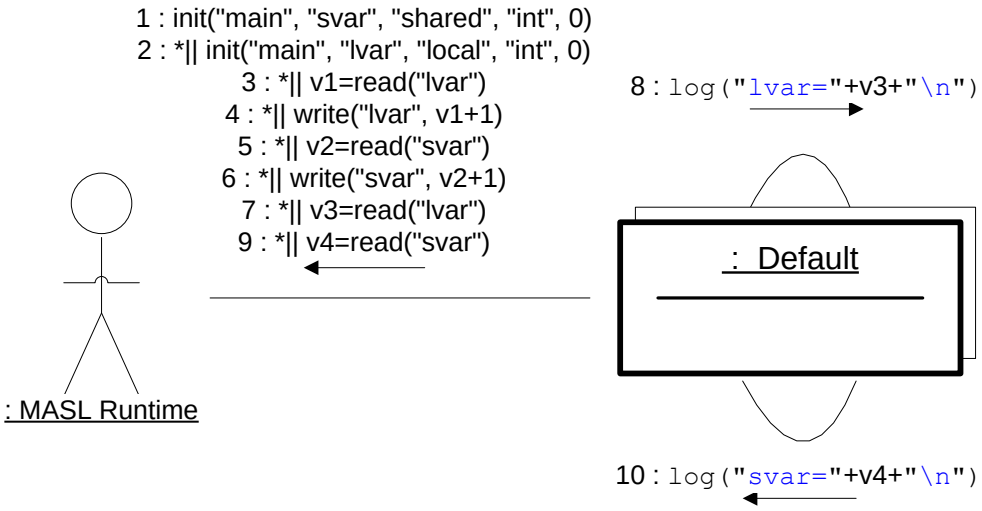
Ces possibilités correspondent à une exécution du programme qui donne un état cohérent de `svar` au final.

Dans le cas où l'affectation `svar++` ne serait pas atomique, donc préemtable (`svar=svar+1` ⇔ `svar+=1`), on pourrait obtenir les exécutions suivantes :

Sixième possibilité	Septième possibilité	Huitième possibilité
lvar= 1 svar= 1 lvar= 1 svar= 1 lvar= 1 svar= 2	Lvar= 1 svar= 1 lvar= 1 svar= 2 lvar= 1 svar= 2	lvar= 1 svar= 1 lvar= 1 svar= 1 lvar= 1 svar= 1

Dans la suite, pour faciliter la tâche du programmeur, nous adopterons une implémentation qui garantie l'atomicité de `svar++`, de `svar=svar+1`. et de `svar+=1`. Incrémenation ou décrémentation sont limitées aux types élémentaires.

Le diagramme de collaboration :



Vision offerte au programmeur :

```
01|
      Tous les agents considérés sont ranges dans un tableau d'agents
      Default (ancêtre de toutes les classes d'agents) référencé par
      la variable a.
03|
04| int svar=0;
05| int[] lvar=new int[main.count];
06| java.util.Arrays.fill(lvar,0);
07| for (i=0 ; i< main.count; i++)
08|   if (true) {                          // asynchronous entry main
09|     a[i].log("lvar="+lvar[i]+"\n");
10|     a[i].log("svar="+svar) +"\n";
```

```
11|    }
12|
```

Cette traduction en un programme séquentiel correspond au premier affichage qui donne pour la variable `svar` *un état final cohérent. La ligne 6 initialise chaque membres du tableau lvar à 0.*

Remarque :

Par contre, il est difficile d'exclure les pertes de mises à jour dans le programme suivant :

```
03| asynchronous entry main (true) {
04|    shared int svar=0;
05|    local int lvar2;
06|    lvar2=svar;
      …
07|    svar=lvar2+1;
08|    .log("svar="+svar+"\n");
09| }
```

Le problème est aussi présent lorsque l'on utilise une variable partagée `svar1` à la place de `lvar1`. Il est fortement dommageable du point de vue des performances d'exclure l'accès en lecture concurrent au niveau d'une variable partagée.

Si l'accès en écriture est le but d'un accès en lecture, le programmeur devra synchroniser ces agents.

2.3.2 La synchronisation d'agents avec des variables partagées

Sur cet exemple nous montrons comment on peut limiter l'accès d'une porte à un seul agent puis comment on peut les resynchroniser.

Rôle de l'algorithme :

Utilisation d'une variable partagée comme une sémaphore dans une section critique pour éviter un accès concurrent problématique à des ressources.

Principe de l'algorithme :

Le changement de valeur d'une variable partagées autorise ou interdit l'accès en parallèle à une ressouce partagée.

Pseudo code de cet exemple :

```
Instancier 4 agents de type Default
Initialisation d'une variable partagée qui joue le rôle d'une sémaphore
Parmi les agents, un seul va avoir un accès exclusif à une ressource ou à
une section critique. Enfin, il modifiera une sémaphore.
Tous les agents attendent la modification de la sémaphore, pour avoir une
action en parallèle.
```

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3,a4 = newAgent(Default);

03| asynchronous entry main (true) {
04|   shared int ssynchro=0;
05|   scalar entry e0 (true) {
06|     .log("do something…\n");
07|     .log("alone\n");
08|     ssynchro=1;
09|   }
10|   while (ssynchro!=1) {}
11|   .log("ready to do something…\n");
12|   .log("all together\n");
13| }
```

Explication du code

Ligne 1 : Définition d'un `type Default` qui importe son interface du fichier `Default.xml`

Ligne 2 : Déclaration de quatre variables `a1`, `a2`, `a3` et `a4` de `type Default` qui sont instanciées à l'aide du constructeur définit dans le fichier XML.

Ligne 3 : La programmation de la mission des agents se fait par une instruction **asynchronous entry main (true)**.

Ligne 4 : Déclaration de la variable partagée `ssynchro` initialisée à 0. Tous les agents dans le bloc main n'accèderont qu'à un seul exemplaire de cette variable.

Ligne 5 : L'entrée `e0` est scalaire : elle ne laisse entrer qu'un seul agent même si tous sont conforme au test. Ceux qui n'entrent pas passent à la ligne 9 sans attendre du fait de la nature asynchrone du bloc père.

Lignes 6 -7 : un seul agent exécute d'éventuelles actions qui appartiennent à une section critique.

Ligne 8 : accès en écriture de la variable partagée par le même unique agent.

Ligne 9 : boucle obligeant les agents à faire de l'attente active jusqu'à ce que l'agent exécutant le bloc `e0` qui peut être vu comme une section critique. L'agent ayant quitté la section critique ne rentrera jamais dans la boucle.

Ligne 10 : les quatre agents exécutent ces éventuelles commandes de manière plus ou moins décalées et à des rythmes différents.

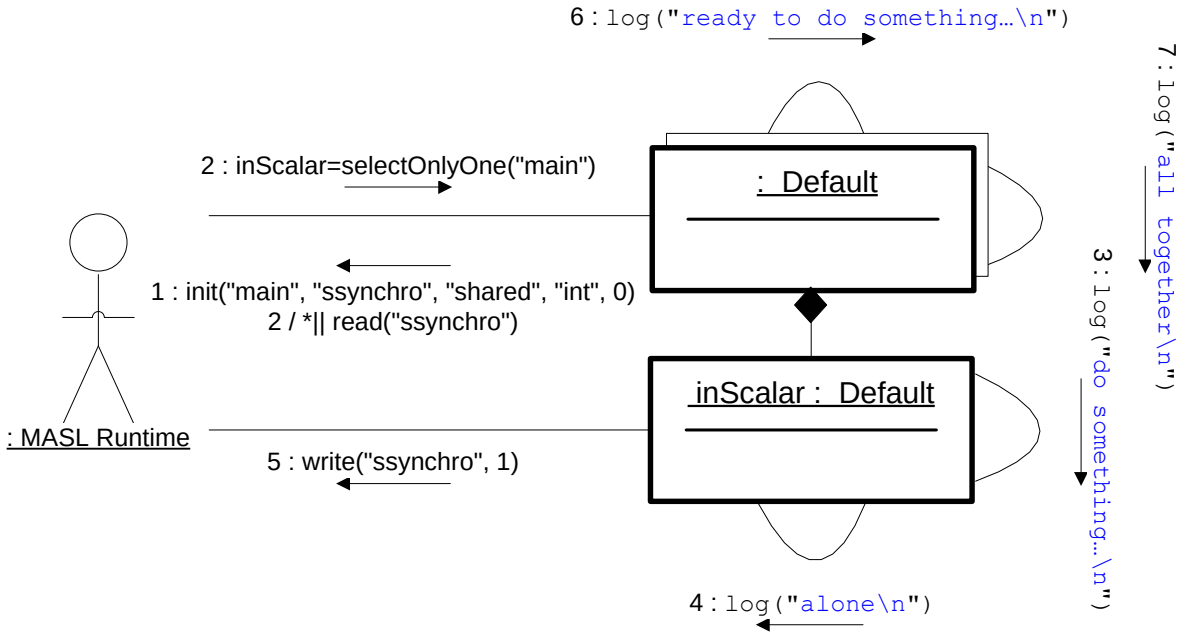
Remarque : On note que la variable `ssynchro` est une variable partagée à l'intérieur du bloc `entry main`.

Affichages possibles du programme dans un seul fichier de journalisation, suivant l'ordonnancement des agents :

Première possibilité	Deuxième possibilité	Autre possibilité
Do something… **alone** ready to do something… all together ready to do something… all together ready to do something… all together ready to do something… all together	**Do something…** **alone** ready to do something… ready to do something… all together all together ready to do something… all together ready to do something… all together	**Do something…** **alone** ready to do something… ready to do something… ready to do something… ready to do something… all together all together all together all together

Il ya plusieurs autres possibilités d'affichage. La seule constante étant l'affichage des deux premières lignes.

Le diagramme de collaboration :



Vision offerte au programmeur :

```
01|
      Tous les agents considérés sont ranges dans un tableau d'agents
      Default (ancêtre de toutes les classes d'agents) référencé par
      la variable a.
03|
04|   int ssynchro=0;
05|   if (true) {                          // scalar entry e0
06|     this.log("do something…\n");
07|     this.log("alone\n");
08|     ssynchro=1;
09|   }
10|   for (i=0 ; i< main.count; i++) {
11|     if (true) {                        // asynchronous entry main
12|       while (ssynchro!=1) {}           // false
13|       a[i].log("ready to do something…\n");
14|       a[i].log("all together\n");
15|     }
16|   }
```

Il s'agit d'un ordonnancement particulier où tous les agents exécutent leur programme lignes par lignes. Cette absence de concurrence n'est pas optimale. Mais cet ordonnancement permet l'exécution correcte de l'algorithme.

2.3.3 Synchronisation et parallélisme synchrone comme alternative à l'utilisation d'une variable de synchronisation

L'exemple précédent montre une attente active. Nous présentons ici une alternative qui s'appuie sur la synchronisation lors de la sortie d'une entrée synchrone. Un bloc `scalar` emboité dans un bloc `synchronous` permet d'obtenir une section critique qui garantie l'accès unique à des ressources gérées dans le bloc scalaire. Le bloc scalaire est considéré comme une seule instruction de la séquence synchrone. L'agent qui l'exécute, le fait en entier. Les autres agents se contentent d'attendre la fin de l'exécution du bloc scalaire.

Rôle de l'algorithme :

Utilisation d'une section critique ou d'un accès exclusif à une ressource.

Principe de l'algorithme :

L'utilisation conjointe d'un bloc scalaire et des synchronisations implicite d'une entrée synchrone permet de gérer une section critique..

Pseudo code de cet exemple :

```
Instancier 4 agents de type Default
Ces agents s'attendent mutuellement pour chaque instruction
Parmi les agents, un seul va avoir un accès exclusif à une ressource ou à
une section critique.
Tous les autres agents attendent la terminaison de la section critique,
pour avoir une action synchronisée en parallèle avec le premier agent.
```

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3,a4 = newAgent(Default);

03| synchronous entry main (true) {
04|   scalar entry e0 (true) {
05|     .log("do something…\n");
06|     .log("alone\n");
07|   }
08|   .log("ready to do something…\n");
09|   .log("all together\n");
10| }
```

Explication du code :

Ligne 3 : Les quatre agents entrent par le bloc main et s'attendent mutuellement après chaque instruction.

Lignes 4-7 : Le bloc scalaire sera exécuté sans aucune attente entre les instructions. Ce bloc emboîté est considéré par le bloc père synchrone comme une seule instruction. Les autres agents, puisqu'ils sont conformes au test, attendent la terminaison de la section critique.

Ligne 8 : Les quatre agents `a1`, `a2`, `a3`, `a4` exécutent `log("ready to do something…\n")` puis s'attendent mutuellement.

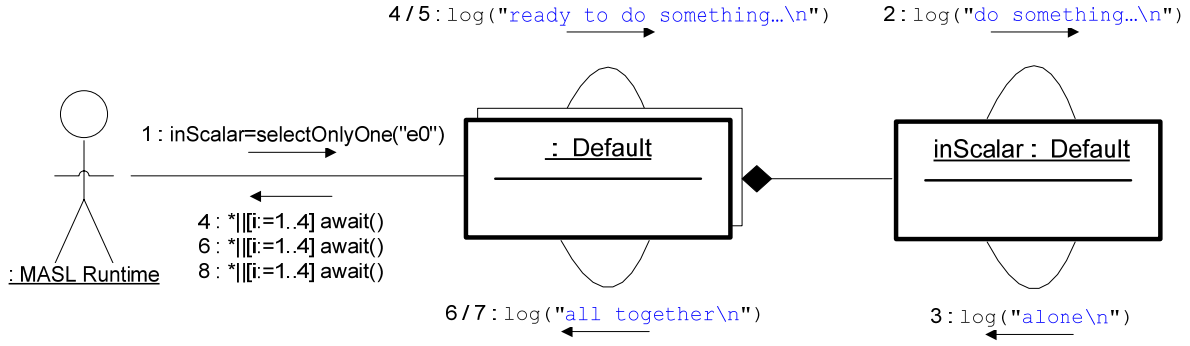
Ligne 9 : Les quatre agents `a1`, `a2`, `a3`, `a4` exécutent `log("all together\n")` puis s'attendent mutuellement.

Affichage du programme dans un seul fichier de journalisation :

Une seule possibilité
Do something… alone ready to do something… ready to do something… ready to do something… ready to do something… all together all together all together all together

Il y a une seule possibilité.

Le diagramme de collaboration :



Vision offerte au programmeur :

```
01|
      Tous les agents considérés sont ranges dans un tableau d'agents
      Default (ancêtre de toutes les classes d'agents) référencé par
      la variable a.
03|
04|   if (true) {                          // scalar entry e0
05|     this.log("do something…\n");
06|     this.log("alone\n");
07|   }
08|   for (i=0 ; i< main.count; i++) {
09|     if (true) {                        // synchronous entry main
10|       a[i].log("ready to do something…\n");
11|   for (i=0 ; i< main.count; i++) {
12|     if (true) {                        // synchronous entry main
13|       a[i].log("all together\n");
```

2.3.4 Synchronisation par évènements

Rôle de l'algorithme :

Première illustration des évènements.

Principe de l'algorithme :

Emission, interruption du flot de contrôle (instruction react), reprise du flot de contrôle (instruction resume), abandon du bloc suite au traitement d'un évènement.

Pseudo code de cet exemple :

```
Instancier 4 agents de type Default
Parmi les agents, un seul va avoir un accès exclusif à une ressource ou à
une section critique. Tous les autres agents attendent la terminaison de la
section critique sanctionnée par un évènement.
D'autres évènements permettent une attente mutuelle des agents.
```

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3,a4 = newAgent(Default);

03| asynchronous entry main (true) {
04|   shared event havestartedmywork;
05|   shared event havefinishedmywork;
06|   shared event waiting;
07|   local int lcounter=0;

08|   scalar entry e0 (true) {
09|     emitWithMessage(havestartedmywork, .getProxyID());
10|     while (lcounter<main.count-1) {}
11|     .log("do something alone\n");
12|     emit(havefinishedmywork);
13|     react (havestartedmywork) {
14|       resume;
15|     }
16|     react (waiting) {
17|       lcounter++;
18|       resume;
19|     }
20|     react (havefinishedmywork) {
21|       .log("done\n");
22|     }
23|   }
24|   while (lcounter==0) {}
25|   .log("ready to do something…\n");
26|   .log("all together\n");
27|   react (havestartedmywork) {
28|     .log("waiting…"+ getMessageFromEvent(havestartedmywork)+"\n");
29|     emit (waiting);
30|     resume;
31|   }
```

```
32|    react (havefinishedmywork) {
33|      lcounter++;
34|      resume;
35|    }
36|    react (waiting) {
37|      resume;
38|    }
39|  }
```

Explication du code :

Lignes 4-6 : Déclaration des évènements partagés : tous les agents présents dans le bloc main en seront destinataires.

Ligne 9 : Un agent émet l'évènement `havestartedmywork` puis attend activement que sa variable `counter` initialement à `0` prenne une certaine valeur. Ne pouvant plus progresser dans son programme, seul le traitement d'un évènement reçu peut le faire réagir.

Ligne 13-15 : L'agent ignore l'évènement `havestartedmywork` qu'il vient d'émettre puisque sa réaction ne se limite qu'à l'instruction **resume** qui lui permet de reprendre son flot de contrôle là où il en était.

Lignes 16-19 : L'attente active ne prendra fin que lorsqu'il aura traité 3 fois l'évènement `Waiting` suite à 3 réceptions. A la fin de la réaction, l'agent reprend son flot de contrôle dans le bloc `e0` là ou il en était avant du fait de l'instruction **resume**.

Lignes 20-22 : Suite à la réception de l'évènement `havefinishedmywork`, l'agent écrit dans le fichier de journalisation et quitte le bloc `e0` du fait de l'absence de l'instruction `resume`.

Ligne 24 : Ce même agent ayant sa variable locale non égale à 0 n'attend pas contrairement aux autres.

Lignes 27-31 : Ces derniers réagissent à l'évènement `havestartedmywork` par l'emission de l'évènement `waiting`, ce qui contribue à libérer le premier de son attente (cf. ligne 10). A noter qu'ils récupèrent un message de l'évènement à réagir qu'ils utilisent lors de l'écriture dans le fichier de journalisation.

Lignes 32-35 : La réaction à l'évènement `havefinishedmywork` les libère de leur attente ligne 24.

Ligne 36 : Les agents, à ce stade, ignorent l'évènement `waiting`, car s'ils se trouvent dans ce bloc, ils ne sont pas la cible principale de cet évènement. Cette dernière est faite des agents suceptibles de réagir ligne 16.

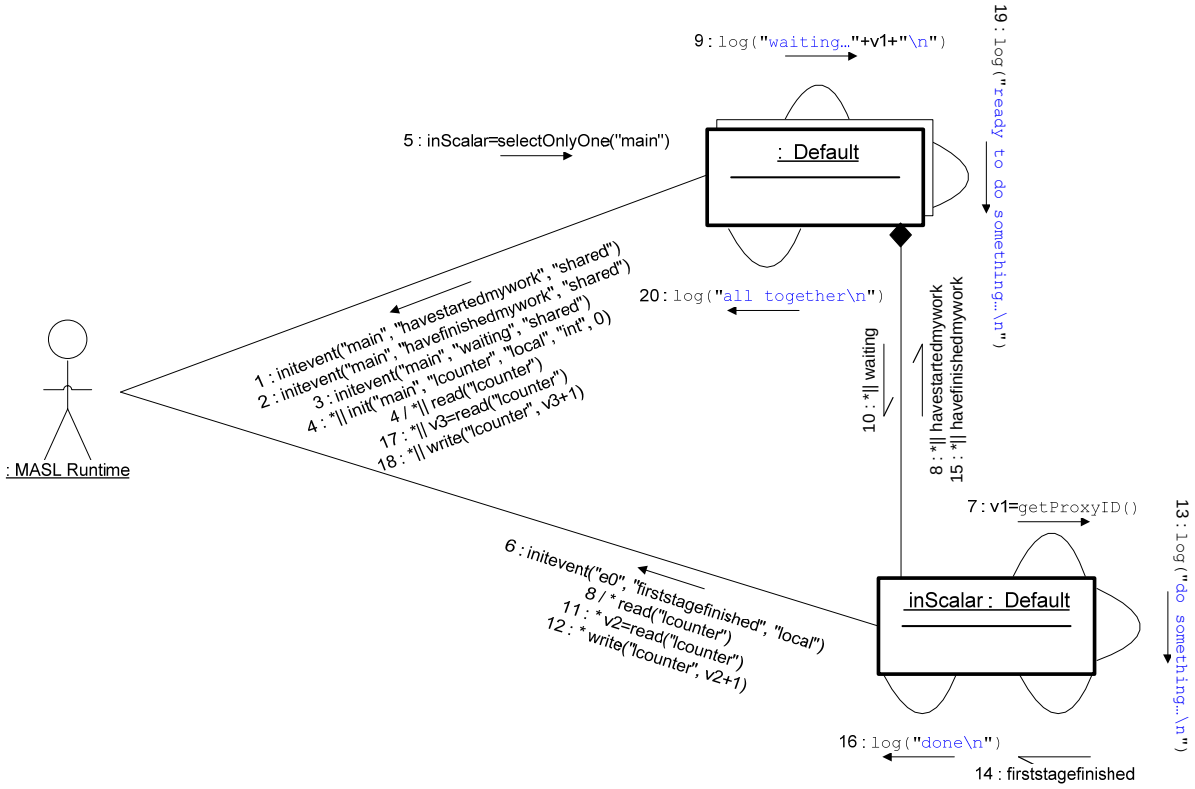
Affichage(s) possible(s) du programme dans un seul fichier de journalisation :

Il faut noter qu'il peut s'insérer un affichage entre « `Do something alone` » et « `Done` ». En effet à l'émission de l'évènement `havefinishedmywork`, l'agent dans `e0` réagit en affichant « `Done` », les autres réagissent en incrémentant une variable qui les fait sortir de l'attente active. On ne peut pas prédire quelle réaction sera la plus rapide.

Première possibilité	Deuxième possibilité	Troisième possibilité
Waiting…0	waiting…1	waiting…2
waiting…0	waiting…1	waiting…2
waiting…0	waiting…1	waiting…2
Do something alone	Do something alone	Do something alone
Done	Done	Done
ready to do something…	ready to do something…	ready to do something…
all together	all together	all together
ready to do something…	ready to do something…	ready to do something…
all together	all together	all together
ready to do something…	ready to do something…	ready to do something…
all together	all together	all together
ready to do something…	ready to do something…	ready to do something…
all together	all together	all together

Quatrième possibilité	Cinquième possibilité	…
Waiting…3 waiting…3 waiting…3 Do something alone Done ready to do something… all together ready to do something… all together ready to do something… all together ready to do something… all together	waiting…0 waiting…0 waiting…0 Do something alone Done ready to do something… ready to do something… all together all together ready to do something… all together ready to do something… all together	

Le diagramme de collaboration :



Vision du programmeur :
La vision du programmeur est difficile à représenter puisque la gestion des évènements implique l'interruption du flot de contrôle. Tout ce que le programmeur doit savoir, c'est qu'après chaque « ; », les évènements reçus par l'agent sont traités : l'agent saute à la réaction de l'évènement reçu. Les évènements sont traités de manière prioritaire et complète. En absence d'instruction **resume**, à la fin de la réaction, l'agent quitte le bloc courant. Dans le cas contraire, il reprend là ou il en était.

2.3.5 Pseudo parallélisme par évènements locaux

Rôle de l'algorithme :
Utilisation des évènements locaux. Seul l'agent émetteur perçoit un évènement local. Il permet de faire communiquer des composants internes à l'agent.

Principe de l'algorithme :
Programmation du fonctionnement interne d'une horloge. On fait l'hypothèse que la fonction `getProxyID` renvoie des nombres qui se suivent, ce qui n'est pas forcément garanti dans la réalité.

Pseudo code de cet exemple :

```
Instancier 4 agents de type Default.
Seule la progression en heure et en jour utilise des évènements locaux.
```

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3,a4 = newAgent(Default);

03| asynchronous entry main (true) {
04|   local int hour=0;
05|   local int second=0;
06|   local event anotherhour;
07|   local event anotherday;
08|   hour+=.getProxyID()%4; /* GMT+0 or GMT+1 or GMT+2 or GMT+3*/
09|   while (true){
10|     Thread.sleep(1000); /* delay= 1 second */
11|     second++;
12|     if (second==60) emit(anotherhour);
13|     if (hour==24) emit(anotherday);
14|   }
15|   react (anotherhour) {
16|     hour++;
17|     second=0;
18|     resume;
19|   }
20|   react (anotherday) {
21|     hour=0;
22|     resume;
23|   }
```

Explication du code :
Ligne 3 : Les quatre agents entrent par le bloc main pour agir de manière désynchronisée.
Lignes 4-5 : Les variables locales `hour` et `second` sont initialisées.
Lignes 6-7 : Les évènements `anotherhour` et `anotherday` sont locaux : seuls les agents qui les émettent les reçoivent.
Ligne 8 : Les agents représentent le temps selon des fuseaux horaires différents. A noter qu'il n'y a aucune garantie que les horloges soient synchronisées
Ligne 10 : L'agent attend une seconde.

2.4 Renforcement de l'équipe par rattachement dynamique d'un agent à une mission en cours

Dans la robotique collective, c'est la coopération de plusieurs agents qui permet l'accomplissement de la mission. Des agents appartiennent à un groupe pour réaliser une tâche. Au sein de ce groupe, la tâche peut nécessiter plusieurs rôles qui sont attribués aux membres du groupe. Ainsi la réalisation des tâches entraîne la définition dynamique des groupes. Si un agent change de rôle au sein d'un groupe, il peut être amené à changer de groupe. Par exemple, si un robot entre en mode dégradé qui l'empêche d'accomplir une tâche, un autre robot peut être assigné à cette dernière. Dans ce cas particulier, la redondance des agents contribue à la tolérance aux pannes du système. Notre proposition permet d'attribuer de manière dynamique une tâche à un agent en fonction de ses capacités et de son état. A un moment donné, il est possible de revenir sur l'opportunité de la tâche en cours en revérifiant ces critères.

2.4.1 La mission de chaque groupe par succession et emboîtements d'entrées, cas du REELECT

Principe de l'algorithme :

Poursuite itérative conditionnée d'un traitement générant des vagues successives d'agent dans des bloc entry.

Pseudo code de cet exemple :

```
Instancier 4 agents de type Default
Lors du travail en équipe, pour chaque itération, un leader va effectuer un
travail supplémentaire, ce qui va l'obliger de se recharger à l'itération
suivante et à subir une révision au bout de 2 périodes de leadership.
```

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3,a4 = newAgent(Default);
03| Entry dostep;

04| asynchronous entry main (true) {
05|   local int ltimetocheck=0;
06|   local int ltimetoload=0;
07|   local int lstepentered=0;
08|   shared boolean sstilltowork=true;
09|   shared int sstep=0;
10|   shared final int sobjectivestep=10;
11|   asynchronous entry checkAndRepair (sstilltowork && ltimetocheck==2) {
12|     lstepentered=sstep;
13|     while (sstilltowork && sstep<lstepentered+2) {
14|     }
15|     ltimetocheck=0;
16|   }
17|   asynchronous entry workinteam (sstilltowork && ltimetocheck<>2) {
18|     asynchronous entry reload (sstilltowork && ltimetoreload==1) {
19|       lstepentered=sstep;
20|       while (sstilltowork && sstep<lstepentered+1) {
21|       }
22|       ltimetoload=0;
23|     }
24|     while (dostep.count>0) {}
25|     asynchronous entry dostep (sstilltowork) {
26|       scalar entry e0 (true) {
27|         .log("do an additional work\n");
28|          ltimetoload++;
29|          ltimetocheck++;
30|       }
31|       .log("ready to do something…in parallel\n");
32|       if (ltimetoload==1) {
33|         if (sstep==sobjectivestep) {
34|           sstilltowork=false;
35|         }
36|         sstep++;
37|         if (ltimetocheck==2) {
38|           reelect(main);
39|         }
40|         reelect(workinteam);
41|       }
```

```
42|         reelect;
43|       }
44|     }
45|  }
```

Explication du code :
Ligne 3 : Le bloc `dowork` est déclaré. Ceci est nécessaire puisque l'on va utiliser dans le code ses informations dynamiques. A noter que par défaut, seul le boc main (obligatoire) est déclaré.
Lignes 5-10 : Il y a initialisation des variables suite à leur déclaration. Toutes ces variables ne seront réinitialisées que si l'entrée où elles ont été définies se retrouve vide. La réinitialisation ne s'opèrera que pour une autre vague d'agent. Dans le cas présent, c'est impossible : lorsque tous les agents auront quitté le bloc main, la mission sera terminée. L'effet obtenu est que chaque agent conserve sa mémoire locale et partagée.
Lignes 11-16: Pour peu qu'il reste du travail et que l'agent doit être en révision, il reste au garage pendant deux itérations.
Lignes 18-23: Pour peu qu'il reste du travail et que l'agent doit être rechargé, il se branche au chargeur pendant une itération.
Ligne 24 : Une entrée scalaire dans une entrée père ne laisse exécuter qu'un seul agent éligible. Si le bloc père est asynchrone, les autres éligibles passent à la suite sans attendre : ils se comportent ainsi comme les non éligibles. Un bloc scalaire ne pourra être exécuté à nouveau qu'après que l'entrée père soit devenue vide, c'est-à-dire que la précédente vague des agents l'ait quittée. Ainsi, avant que la nouvelle vague ne rentre, il faut que les retardataires l'aient quittée. Avant que les agents entre dans le bloc main, les structures de données décrivant les blocs entry sont initialisée. Tant qu'aucun agent ne rentre dans le bloc dostep, l'instruction dostep.count renvoie la valeur 0. Ainsi compte tenu des conditions initiales, le premier agent n'attendra pas.
Lignes 25-30 : Le leader de l'itération fait son travail supplémentaire et inscrit dans son état qu'il doit être rechargé durant la prochaine itération. De plus, son état prend en compte son droit à une révision prochaine.
Ligne 31 : Tous les agents impliqués dans l'itération de la tache collective font leur part du travail.
Lignes 32-41 : le code ne concerne que l'agent actuellement leader. Il est unique donc il peut incrémenter sans problème `sstep` ligne 36.
Ligne 38 : L'agent actuellement leader va devoir vérifier que son état est conforme aux conditions d'entrée du bloc `main`.
Ligne 40 : L'agent actuellement leader qui disposait déjà d'un droit à révision va devoir vérifier que son état est conforme aux conditions d'entrée du bloc `workinteam`.
Ligne 42 : L'agent va devoir vérifier que son état est conforme aux conditions d'entrée du bloc `dostep`.

2.4.2 La mission de chaque groupe par succession et emboîtements d'entrées, cas de RESTART

Rôle de l'algorithme :

Utilisation conjointe des évènements et de restart. Ceci permet d'illustrer des tâches interruptibles qui se terminent lorsqu'une tâche non interruptible termine normalement. Pour cela, on utilise un évènement `abort`.

Principe de l'algorithme :

L'utilisation conjointe d'un bloc scalaire et des évènements permet de gérer une section critique.

Pseudo code de cet exemple :

```
Instancier 4 agents de type Default
Suite à une tache d'initialisation par un agent, la population fait un
travail interruptible.
```

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3,a4 = newAgent(Default);
03| Entry restartable;

04| synchronous entry main (true) {
05|   scalar entry e0 (true) {
06|     .log("do initialization work once\n");
07|   }
08|   asynchronous entry dowork (true) {
09|     shared int sstep=0;
10|     shared final int sobjectivestep=10;
11|     local boolean lleader;
12|     shared event abort;
13|     lleader=false;
14|     asynchronous entry restartable (true) {
15|       scalar entry e1 (true) {
16|         .log("do an additional work\n");
17|         lleader=true;
18|         react (abort) {
19|           break(dowork);
20|         }
21|       }
22|       .log("ready to do something…in parallel\n");
23|       if (lleader) {
24|         lleader=false;
25|         if (sstep==sobjectivestep) {
26|           emit(abort);
27|         }
28|         sstep++;
29|       }
30|     }
31|     while (restartable.count!=0) {};
32|     restart;
33|     react (abort) {
34|       break;
35|     }
36|   }
```

```
37 | }
```

Explication du code :
Ligne 3 : Le bloc `restartable` est déclaré. Ceci est nécessaire puisque l'on va utiliser dans le code ses informations dynamiques. A noter que par défaut, seul le boc main (obligatoire) est déclaré.
Lignes 5-7 : Une entrée scalaire dans une entrée père ne laisse exécuter qu'un seul agent éligible. Comme l'entrée père est synchrone, les autres éligibles attendent la fin de l'exécution du bloc pour être libérés avec celui qui l'a exécuté. Un bloc scalaire ne pourra être exécuté à nouveau qu'après que l'entrée père soit devenue vide, c'est-à-dire que la précédente vague des agents l'ait quittée. Ici, c'est impossible car lorsque tous les agents quittent l'entrée main, la mission est terminée.
Ligne 9 : la variable `sstep` ne sera jamais réinitialisée et elle sera conservée lors de chaque itération.
Ligne 10 : la constante ne peut être initialisée qu'une fois.
Ligne 13 : la variable `lleader` sera systématiquement réaffectée à chaque passage de l'agent.
Lignes 15-21 : Un bloc scalaire ne pourra être exécuté à nouveau qu'après que l'entrée père soit devenue vide, c'est-à-dire que la précédente vague des agents l'ait quittée. Ici, c'est la boucle ligne 31 qui garantit que le bloc `restartable` est vide, ce qui réactive le bloc e1 pour la prochaine vague.
Ligne 31 : On attend que tous les agents quittent le bloc `restartable`. L'information utilisée est le nombre d'agent présent dans l'entrée.
Ligne 32 : l'emploi de l'instruction restart fait que l'agent recommence à traiter l'instruction de la ligne 8 sans vérification des conditions d'entrée.
Ligne 34 : Le traitement itératif sera interrompu par le traitement de l'évènement `abort`.

2.4.3 La dynamique d'un bloc synchrone dans une seule passe

Rôle de l'algorithme :
Il s'agit d'illustrer l'emboitement dans un bloc synchrone d'une séquence de blocs synchrone et asynchrones. Les agents ne font pas de retour arrière. Ainsi, les agents changent de rythme en cours d'exécution. On fait l'hypothèse que la fonction `getProxyID` renvoie des nombres qui se suivent, ce qui n'est pas forcément garanti dans la réalité.

Principe de l'algorithme :
L'écriture de ligne continue de "-" n'est pas sensible à l'ordre des caractères, ce qui n'est pas le cas de l'écriture des mots. Aussi va-t-on employer, dans cet exemple, l'écriture par des agents synchronisés de parties de mots dans un tampon. L'écriture de la ligne de "-" se fait directement par des agents désynchronisés.

Pseudo code de cet exemple :

```
Instancier 4 agents de type Default
Ces agents sont synchronisés pour écrire les parties des mots dans un
tampon.
Puis ils écrivent directement à leur rythme la ligne continue.
Un seul écrira le retour à la ligne terminal.
```

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3,a4 = newAgent(Default);

03| synchronous entry main (true) {
04|   shared String sline;
05|   local String lmessage="";
06|   scalar entry initbuffer (true) {
07|     sline="            ";
08|   }
09|   if (.getProxyID() %2==0) lmessage="hip";
10|   else lmessage="hop";
11|   sline=sline.substring(0,.getProxyID()*3)+lmessage+
      sline.substring((.getProxyID()+1)*3);
12|   scalar entry printbuffer (true) {
13|     .log(sline+"\n");
14|   }
15|   asynchronous entry minuslinewithoutbuffer (.getProxyID()%2==0) {
16|     .log("-");
17|     .log("-");
18|     .log("-");
19|     .log("-");
20|   }
21|   .log("**");
22|   scalar entry carriagereturnonlywithoutbuffer (true) {
23|     .log("\n");
24|   }
25| }
```

Explication du code :
Ligne 7 : Un tampon est initialisé à la bonne taille.
Ligne 11 : L'agent replace sa partie du buffer par la succession des trois lettres qu'il doit écrire. Deux méthodes de la classe `java.lang.String` sont employées : la première renvoie la sous-chaîne précisée par son début et sa longueur. La deuxième renvoie la sous-chaîne à partir de son unique paramètre qui précise la position de début.

Ligne 13 : Parmi le bloc synchrone, un seul agent écrit le tampon dans le fichier de journalisation. Pendant ce temps les autres agents l'attende du fait du caractère synchrone du bloc père.
Ligne 15-20 : Deux agents exécutent le bloc asynchrone à leur rythme. Ils s'attendront à la fin du bloc qui est considéré par le bloc synchrone père comme une seule instruction. Ils se contentent d'écrire quatre mêmes caractères dans le même fichier de journalisation.
Ligne 21 : Les quatre agents écrivent deux mêmes caractères dans le même fichier de journalisation puis s'attendent mutuellement.
Ligne 22-24 : Un seul agent écrit le retour à la ligne après que toute la ligne de "−" et de "⋆" ait été écrite. Les autres l'attendent.

Affichage(s) possible(s) du programme dans un seul fichier de journalisation :

Une seule possibilité
Hiphophiphop −−−−−−−−⋆⋆⋆⋆⋆⋆⋆⋆

2.4.4 La dynamique d'un bloc synchrone : distinction premier et autre passage d'agents

Rôle de l'algorithme :
Il s'agit d'illustrer l'emboitement dans un bloc synchrone d'une séquence de blocs synchrone et asynchrones. Plusieurs vagues d'agents empruntent le bloc synchrone. La politique d'activation de la première vague diffère des suivantes par défaut. On fait l'hypothèse que la fonction `getProxyID` renvoie des nombres qui se suivent, ce qui n'est pas forcément garanti dans la réalité.

Principe de l'algorithme :
L'écriture de ligne continue de "-" n'est pas sensible à l'ordre des caractères, ce qui n'est pas le cas de l'écriture des mots. Aussi va-t-on employer, dans cet exemple, l'écriture par des agents synchronisés de parties de mots dans un tampon. L'écriture de la ligne de "-" se fait directement par des agents désynchronisés.

Pseudo code de cet exemple :

```
Instancier 4 agents de type Default
Ces agents sont synchronisés pour écrire les parties des mots dans un
tampon.
Puis ils écrivent directement à leur rythme la ligne continue.
Un seul écrira le retour à la ligne terminal.
```

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3,a4,a5,a6 = newAgent(Default);
03| Entry write;

04| asynchronous entry main (true) {
05|   synchronous entry dowork (true) {
06|     shared String sline;
07|     local String lmessage="";
08|     asynchronous entry other (.getProxyID()<4) {
09|       .log("doing a small work\n");
10|     }
11|     synchronous entry write (.getProxyID()>=2) {
12|       shared int sindex;
13|       local int lindex;
14|       scalar entry init (true) {
15|         sline="                    ";
16|         sindex=0;
17|       }
18|       lindex=sindex++;
19|       if (.getProxyID()==2) lmessage="hip";
20|       else if (.getProxyID()==3) lmessage="hop";
21|       else if (.getProxyID()==4) lmessage="tic";
22|       else lmessage="tac";
23|       sline=sline.substring(0, lindex *3)+lmessage+
24|       sline.substring((lindex +1)*3);
25|       scalar entry print (true) {
26|         .log(sline+"\n");
27|       }
28|     }
29|     if (write.getWaitingCount()>0 {
30|       while (write.getActiveCount()!=0) {};
31|       write.release();
32|     }
```

```
33|     }
34| }
```

Explication du code :

Ligne 3 : Le bloc `write` est déclaré. Ceci est nécessaire puisque l'on va utiliser dans le code ses informations dynamiques. A noter que par défaut, seul le boc `main` (obligatoire) est déclaré.

Lignes 8-10 : Le bloc asynchrone capture les quatre premiers agents qui de fait seront dissociés par rapport à aux 2 autres qui rentrent immédiatement dans le bloc synchrone suivant (11-28). Certes, la barrière de synchronisation du bloc `dowork` oblige les quatre et les deux autres à s'attendre à la fin de la ligne 10 et17. Mais le test ligne 11 des deux derniers agents a fait basculer la politique d'entrée du bloc write de `free` à `locked` : seul l'emploi de l'instruction `write.release()` peut faire s'exécuter les agents élus dans le bloc. Il faut noter qu'un seul des deux agents exécutera les lignes 15 et 16 et que l'autre l'attend.

Lignes 10 : Les deux premiers agents (0 et 1) quittent le bloc `dowork`.

Lignes 11 à 28 : le traitement est identique à l'exemple précédent : une écriture synchronisée dans un tampon. L'originalité, c'est qu'il y aura 2 vagues d'agents à parcourir ce bloc synchrone. Les agents 2 et 3 sont en retard par rapport aux agents 4 et 5. Ces derniers rentrent sans attente. Les agents 2 et 3 devront attendre que la première vague ait quitté le bloc et qu'un agent les autorise à entrer dans le bloc. Cela peut être un des agents 0 et 1 ou un des agents de la première vague (4 ou 5) qui va le faire à la ligne 31.

2.4.5 La dynamique d'un bloc synchrone : gestion de l'activation du bloc

Rôle de l'algorithme :
Il s'agit d'illustrer l'emboitement dans un bloc synchrone d'une séquence de blocs synchrone et asynchrones. Plusieurs vagues d'agents empruntent le bloc synchrone. La politique d'activation de la première vague est redéfinie.

Principe de l'algorithme :
Les informations des blocs entry sont mis a profit pour court-circuiter la politique par défaut de libération des portes à la première vague.

Pseudo code de cet exemple :

```
Instancier 4 agents de type Default
Pour garantir que la politique de libération est redefinie avant que tout
agent soit attribué aux blocs, on recourt à un bloc scalar et à une attente
active.
```

Code MASL correspondant :

```
01| import Default.xml as Default;
02| Default a1,a2,a3,a4,a5 = newAgent(Default);
03| Entry dowork;

04| asynchronous entry main (true) {
05|   scalar entry super (true) {
06|     dowork.lock();
07|     while (main.getActiveCount()>1) {
08|       if (dowork.getActiveCount()==0 && dowork.getWaitingCount()>=2)
09|       dowork.release(2);
10|     }
11|   }
12|   while (dowork.getLockPolicy()=="free") {
13|   }
14|   while (main.getActiveCount()>1) {
15|     synchronous entry dowork (true) {
16|       shared int slock=0;
17|       local int llock=slock++;
18|       if (llock==0) {
19|         .log("hip\n");
20|         .nop();
21|       else  {
22|         .nop();
23|         .log("hop\n");
24|       }
25|     }
26|   }
27| }
```

Explication du code :
Ligne 3 : Le bloc `dowork` est déclaré. Ceci est nécessaire puisque l'on va utiliser dans le code ses informations dynamiques et ses méthodes.
Lignes 6 et 12-13 : Il s'agit d'empêcher la libération automatique des agents une fois qu'ils ont été élus dans le bloc `dowork`.
Lignes 7-10 : un agent superviseur gère l'ouverture de la porte du bloc `dowork`. Cette dernière libère seulement 2 agents.
Lignes 14 à 26 : Pour peu qu'il reste d'autres agents que le superviseur, un traitement en boucle nécessitant deux agents s'opère.

Ligne 16 : A chaque vague de 2 agents, la variable `slock` est réinitialisée.
Ligne 20 : l'instruction `nop()` ne fait rien. Il s'agit d'un service de base de tous les agents.

2.4.6 Envoi de messages synchrones et asynchrones

L'API des agents peuvent classiquement comporter des méthodes bloquantes. De plus, l'agent peut offrir un parallélisme interne grâce à des méthodes non bloquantes.

2.4.6.1 Envoi de messages synchrones

```
01| import Default.xml as Default;
02| Default leader,a1,a2,a3 = newAgent(Default);

03| asynchronous entry main (true) {
04|   LLabel llabel;
05|   local int lleaderID;
06|   local int lID;
07|   llabel.lleaderID=leader.getProxyID();
08|   lID=.getProxyID();
09|   llabel.log("Calls of synchronous methods\n");
10|   while (!isFinished(llabel)){
11|     .log("Waiting 1 second…\n");
12|     Thread.sleep(1000);
13|   }
14|   .log("---\n");
15|   .log("["+lID+"] "] The leader ID is: "+lleaderID+"\n");
16| }
```

Avant de commenter le programme, il faut aussi se referrer à la déclaration des méthodes de l'API pour connaître leur nature synchrone (bloquante pour l'appelant) ou asynchrone (non bloquante pour l'appelant).

Voici donc un extrait du fichier `Default.xml` :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Agent SYSTEM "I:\These\agent.dtd">
<Agent short_class_name="Default" full_class_name="valoria.MASL.Agent.Default">
…
        <Primitive function_name="getProxyID" member_id="ID_1" synchronous_call="true"
return_type_name=" int.class">
                <Modifier modifier_name="public"/>
        </Primitive>
        <Primitive function_name="nop" member_id="ID_2" synchronous_call="true" return_type_name="
void">
                <Modifier modifier_name="public"/>
        </Primitive>
        <Primitive function_name="log" member_id="ID_3" synchronous_call="true" return_type_name="
void">
                <Modifier modifier_name="public"/>
                <Parameter parameter_name="message" parameter_type="java.lang.String"/>
        </Primitive>
…
```

Explication du code :
Ligne 4 : Déclaration d'une étiquette de synchronisation locale.
Ligne 7 : Invocation externe d'une méthode avec attribution d'une étiquette de synchronisation. La méthode invoquée est bloquante.
Ligne 8 : Invocation interne d'une méthode sans attribution d'une étiquette de synchronisation. La méthode invoquée est bloquante.

Ligne 9 : Invocation externe d'une méthode avec attribution de la même étiquette de synchronisation. La méthode invoquée est bloquante.
Ligne 10-13 : La boucle ne sera jamais exécutée puisque l'ensemble des invocations de méthodes effectuées sous l'étiquette de synchronisation ne comportait pas d'élément non bloquant.

Première possibilité	Deuxième possibilité	Troisième possibilité
AcceptState of the leader: default AcceptState of the 0: default [0:User] The leader ID is: 0 AcceptState of the 1: default [1:Group] The leader ID is: 0 AcceptState of the 2: default [2:Others] The leader ID is: 0 AcceptState of the 3: default [3:Others] The leader ID is: 0	AcceptState of the leader: nothing AcceptState of the 0: nothing [-1:User] The leader ID is: -1 AcceptState of the 1: default [1:Group] The leader ID is: -1 AcceptState of the 2: default [2:Others] The leader ID is: -1 AcceptState of the 3: default [3:Others] The leader ID is: -1	AcceptState of the leader: profile2 AcceptState of the 0: profile2 [0:User] The leader ID is: 0 AcceptState of the 1: default [1:Group] The leader ID is: -1 AcceptState of the 2: default [2:Others] The leader ID is: -1 AcceptState of the 3: default [3:Others] The leader ID is: -1

Quatrième possibilité
AcceptState of the leader: profile1 AcceptState of the 0: profile2 [0:User] The leader ID is: 0 AcceptState of the 1: default [1:Group] The leader ID is: 0 AcceptState of the 2: default [2:Others] The leader ID is: -1 AcceptState of the 3: default [3:Others] The leader ID is: -1

D'autres affichages sont possibles.

2.4.6.2 Envoi de messages asynchrones

```
01| import Default.xml as Default;
02| Default leader,a1,a2,a3 = newAgent(Default);

03| asynchronous entry main (true) {
04|   LLabel llabel;
05|   local int lleaderID;
06|   local int lID;
07|   llabel.lleaderID=leader.getfProxyID();
08|   lID=.getfProxyID();
09|   llabel.log("Calls of synchronous methods\n");
10|   while (!llabel.isFinished()){
11|     .log("Waiting 1 second…\n");
12|     Thread.sleep(1000);
13|   }
14|   .log("---\n");
15|   .log("["+lID+"] "] The leader ID is: "+lleaderID+"\n");
16| }
```

```
…
        <Primitive function_name="getfProxyID" member_id="ID_4" synchronous_call="false"
return_type_name=" int.class">
                <Modifier modifier_name="public"/>
        </Primitive>
…
```

Explication du code :
Ligne 7 : Invocation externe d'une méthode avec attribution d'une étiquette de synchronisation. La méthode invoquée est non bloquante.
Ligne 8 : Invocation interne d'une méthode sans attribution d'une étiquette de synchronisation. La méthode invoquée est non bloquante.
Ligne 9 : Invocation externe d'une méthode avec attribution de la même étiquette de synchronisation. La méthode invoquée est bloquante.
Ligne 10-13 : La boucle peut être exécutée puisque l'ensemble des invocations de méthodes effectuées sous l'étiquette de synchronisation comportait des éléments non bloquant. L'instruction **isfinished** peut donc être évalué à `false`.
Ligne 15 : Il peut y avoir une attente jusqu'à ce que le résultat de l'appel ligne 8 soit disponible dans `lID` puisque cet appel n'était pas inclus dans l'étiquette de synchronisation.

2.4.6.3 Envoi de messages asynchrones avec contre-mesures

Un robot peut nécessiter de passer par des états transitoires pour ne pas l'endommager. La terminaison prématurée d'une méthode de l'API peut être préjudiciable. Une action de compensation est peut être nécessaire. Dans le cadre d'une communication asynchrone, l'appel implicite de méthode de correction de l'état du robot suite à une terminaison prématurée garantit l'intégrité du robot.

```
01| import Default.xml as Default;
02| Default a0,a1,a2,a3 = newAgent(Default);
03| Entry critical;

04| asynchronous entry main (true) {
05|   .setfSomethingDangerousOn();
06|   .log("Doing something\n");
07|   if (.getProxyID()==2) break;
08|   synchronous entry critical (true) {
09|     .setfSomethingVeryDangerousOn();
10|     .log("Doing something\n");
11|     if (.getProxyID()==0) break;
12|     if (.getProxyID()==1) break(main);
13|     .log("Doing something\n");
14|     .setfSomethingVeryDangerousOff();
15|   }
16|   .log("Doing something\n");
17|   .setfSomethingDangerousOff();
18| }
```

```
…
        <Primitive function_name="setfSomethingDangerousOn" member_id="ID_5"
synchronous_call="false" return_type_name=" void" countermeasure=" setfSomethingDangerousOff " >
                <Modifier modifier_name="public"/>
        </Primitive>
        <Primitive function_name="setfSomethingDangerousOff" member_id="ID_6"
synchronous_call="false" return_type_name=" void" >
                <Modifier modifier_name="public"/>
        </Primitive>
        <Primitive function_name="setfSomethingVeryDangerousOn" member_id="ID_7"
synchronous_call="false" return_type_name=" void" countermeasure=" setfSomethingVeryDangerousOff " >
                <Modifier modifier_name="public"/>
        </Primitive>
        <Primitive function_name="setfSomethingVeryDangerousOff" member_id="ID_8"
synchronous_call="false" return_type_name=" void" >
                <Modifier modifier_name="public"/>
        </Primitive>

…
```

Explication du code :
Ligne 3 : L'entrée `critical` est déclarée pour pouvoir être utilisée comme portée d'une contre-mesure.
Ligne 5 : Appel à une méthode asynchrone ayant une contre-mesure. L'appel ne précise pas la portée de la contre-mesure. Par défaut, elle sera invoquée automatiquement lorsque l'agent quitte le bloc main.
Ligne 7 : L'agent n°2 quitte le bloc main. La méthode `setfSomethingDangerousOff` est automatiquement invoquée.
Ligne 9 : Appel à une méthode asynchrone ayant une contre-mesure. La portée de la contre-mesure est le bloc `critical`. Elle sera invoquée automatiquement lorsque l'agent quitte le bloc `critical`.

Ligne 11 : L'agent 0 quitte le bloc critical : la méthode `setfSomethingVeryDangerousOff` est appelée automatiquement.
Ligne 12 : L'agent 1 quitte le bloc main : la méthode `setfSomethingVeryDangerousOff` puis `setfSomethingDangerousOff` sont appelées automatiquement.
Ligne 14 : L'appel à la méthode correspond à la contre-mesure. Ceci à pour conséquence de désactiver pour l'agent l'appel automatique en sortie de bloc `critical`.
Ligne 17 : L'appel à la méthode correspond à la contre-mesure. Ceci à pour conséquence de désactiver pour l'agent l'appel automatique en sortie de bloc main.

2.5 La perméabilité

Lorsque les plateformes sont spécifiques, on voit apparaître des modes dégradés. Des pannes (dysfonctionnement : le moteur du bras ne démarre pas) et la rareté de ressources (énergie faible) empêchent le robot de rendre le service demandé. La *software evolution* permet d'adapter la mission du robot à ses capacités. Les ressources peuvent être évaluées et certaines pannes peuvent être diagnostiquées. Le diagnostic peut être fait soit directement par l'API, soit à l'aide d'un algorithme implanté dans la programmation de la mission. Par exemple un diagnostic global nécessite la prise en compte de l'environnement et du contexte de la mission, ce qui peut ne pas être connu au niveau de l'API. L'API d'un agent peut même proposer différents modes dégradés dans lesquels il peut fonctionner si ces derniers sont facilement identifiables. Selon notre approche, dans un mode dégradé, l'agent ignore certaines sollicitations. La gestion des modes dégradés comprend à la fois le passage d'un mode de fonctionnement à un autre (mode normal vers mode dégradé) suite à un diagnostic, l'évaluation du mode actuel, et la réaction à avoir selon ce dernier afin d'obtenir la tolérance aux pannes. Notre solution propose des mécanismes souples pour prendre en compte la gestion des pannes prévues au niveau de l'API de l'agent ou pour l'implanter soit au niveau du programme de l'agent, ou soit au niveau du groupe. Ainsi les modes dégradés identifiables sont présent dans l'interface XML de l'agent. De plus, les exceptions non capturées de l'API de l'agent sont promues en évènements locaux pour compléter la prise en compte des défaillances envisageables au niveau de la définition de la mission.

Un agent est autonome et peut rejeter les demandes émanant d'autres agents. La coordination d'une équipe peut nécessiter que le niveau d'autonomie d'un agent soit fonction de l'agent lui-même mais aussi de son équipe et en particulier des autres membres. Par exemple, le rejet d'un service peut être fait pour des considérations locales à l'agent (pas assez d'énergie). La motivation du refus peut être aussi par exemple la présence de coéquipiers suffisamment nombreux ou mieux dotés que l'agent. Nous proposons le mécanisme de perméabilité qui gère l'autonomie des agents en fonction de la provenance du message. Un agent prend en compte tout message qu'il reçoit. Sa perméabilité définit s'il ignore une invocation externe : la méthode ne sera pas invoquée. Il ignore rarement une invocation interne. La perméabilité d'un agent peut être définie au niveau du programme local ou de la mission de l'équipe.

2.5.1 Perméabilité pilotée par l'API

```
01| import Default.xml as Default;
02| Default leader,a1,a2,a3 = newAgent(Default);

03| asynchronous entry main (true) {
04|   local int lleaderID;
05|   local int lID=.getProxyID();
06|   .log("AcceptState of the leader: "+getAcceptState(leader)+"\n");
07|   asynchronous entry group (true) {
08|     lleaderID=leader.getProxyID();
09|     react (timeout) {
10|       lleaderID =-1;
11|       resume;
12|     }
13|     lID=.getProxyID();
14|     .log("AcceptState of "+lID+": "+getAcceptState()+"\n");
15|     if (lID<>lleaderID && lID>1) break;
16|     .log("["+lID);
17|     if (lID<>lleaderID) .log(":Group] ");
18|     else .log(":User] ");
19|     .log("The leader ID is: "+lleaderID+"\n");
20|     break (main);
21|   }
22|   asynchronous entry other (true) {
23|     lleaderID=leader.getProxyID();
24|     react (timeout) {
25|       lleaderID =-1;
26|       resume;
```

```
27|       }
28|       .log("["+lID+":Others] The leader ID is: "+lleaderID+"\n");
29|
30|    }
31|    react (timeout) {
32|       lID =-1;
33|       resume;
34|    }
35| }
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Agent SYSTEM "I:\These\agent.dtd">
<Agent short_class_name="Default" full_class_name="valoria.MASL.Agent.Default">
…
<Primitive function_name="getProxyID" member_id="ID_1" synchronous_call="true" return_type_name="
int.class">
<Modifier modifier_name="public"/>
</Primitive>
<Primitive function_name="nop" member_id="ID_2" synchronous_call="true" return_type_name=" void">
<Modifier modifier_name="public"/>
</Primitive>
<Primitive function_name="log" member_id="ID_3" synchronous_call="true" return_type_name=" void">
<Modifier modifier_name="public"/>
<Parameter parameter_name="message" parameter_type="java.lang.String"/>
</Primitive>
…
<Permeability permeability_name="nothing" umask="777"/>
<Permeability permeability_name="default" umask="000">
<Member member_id="ID_1" rights="111"/>
<Member member_id="ID_2" rights="111"/>
<Member member_id="ID_3" rights="111"/>
</Permeability>
<Permeability permeability_name="profile1" umask="000">
<Member member_id="ID_1" rights="110"/>
<Member member_id="ID_2" rights="111"/>
<Member member_id="ID_3" rights="111"/>
</Permeability>
<Permeability permeability_name="profile2" umask="000">
<Member member_id="ID_1" rights="100"/>
<Member member_id="ID_2" rights="111"/>
<Member member_id="ID_3" rights="111"/>
</Permeability>
</Agent>
```

Explication du code :
Ligne 5 et lignes 31 à 34 : Les agents invoquent de manière interne la méthode `getProxyID`. Si cette invocation est ignorée du fait de la perméabilité, l'évènement système timeout est émis. Son traitement est de mettre à -1 la variable `lID`. Ce cas peut se produire si la permeabilité courante est « `nothing` » (interdiction d'exécution pour l'user).
Ligne 6 : L'instruction **getAcceptState** permet à un agent de connaître sa perméabilité courante ou celle d'un autre agent.
Lignes 8 à 12 : Les agents invoquent de manière externe la méthode `getProxyID`. Si cette invocation est ignorée du fait de la perméabilité, l'évènement système timeout est émis. Son traitement est de mettre à -1 la variable `lleaderID`. Ce cas peut se produire si la perméabilité courante est « `nothing` » ou « `profile2` » (interdiction d'exécution pour le groupe)
Lignes 23 à 27 : Les agents invoquent de manière externe la méthode `getProxyID`. Si cette invocation est ignorée du fait de la perméabilité, l'évènement système timeout est émis. Son traitement est de mettre à -1 la variable lleaderID. Ce cas peut se produire si la perméabilité courante est « `nothing` », « `profile1` » ou « `profile2` » (interdiction d'exécution pour `other`).

Première possibilité	Deuxième possibilité	Troisième possibilité
AcceptState of the leader: default AcceptState of the 0: default [0:User] The leader ID is: 0 AcceptState of the 1: default [1:Group] The leader ID is: 0 AcceptState of the 2: default [2:Others] The leader ID is: 0 AcceptState of the 3: default [3:Others] The leader ID is: 0	AcceptState of the leader: nothing AcceptState of the 0: nothing [-1:User] The leader ID is: -1 AcceptState of the 1: default [1:Group] The leader ID is: -1 AcceptState of the 2: default [2:Others] The leader ID is: -1 AcceptState of the 3: default [3:Others] The leader ID is: -1	AcceptState of the leader: profile2 AcceptState of the 0: profile2 [0:User] The leader ID is: 0 AcceptState of the 1: default [1:Group] The leader ID is: -1 AcceptState of the 2: default [2:Others] The leader ID is: -1 AcceptState of the 3: default [3:Others] The leader ID is: -1

Dernière possibilité
AcceptState of the leader: profile1 AcceptState of the 0: profile2 [0:User] The leader ID is: 0 AcceptState of the 1: default [1:Group] The leader ID is: 0 AcceptState of the 2: default [2:Others] The leader ID is: -1 AcceptState of the 3: default [3:Others] The leader ID is: -1

2.5.2 Adaptation de la perméabilité lors de la mission

```
01| import Default.xml as Default;
02| Default printer,battery,a1,a2,a3 = newAgent(Default);
03| Entry large, small, nothing;

04| asynchronous entry main (true) {
05|   local int batterylevel=255;
06|   asynchronous entry toprint (.getProxyID()==printer.getProxyID()) {
07|     local Default[] list=null;
08|     local int l=0;
09|     while (main.getActiveCount()>2 && batterylevel>0) {
10|       list= large.getActiveList();
11|       for (l=0; l<list.length;l++)
12|         .log("[Large:"+list[l].getProxyID()+"] AcceptState: "
          +getAcceptState(list[l])+"\n");
13|       list= small.getActiveList();
14|       for (l=0; l<list.length;l++)
15|         .log("[Small:"+list[l].getProxyID()+"] AcceptState: "
          +getAcceptState(list[l])+"\n");
16|       list= nothing.getActiveList();
17|       for (l=0; l<list.length;l++)
18|         .log("[Nothing:"+list[l].getProxyID()+"] AcceptState: "
          +getAcceptState(list[l])+"\n");
19|       batterylevel= batterylevel-6;
20|     }
21|     break (2);
22|   }
23|   asynchronous entry group (true) {
24|     asynchronous entry large (getAcceptState()=="default") {
25|       Thread.sleep(2000); /* delay= 2 second */
26|       batterylevel= batterylevel-30;
27|       if (batterylevel<180) setAcceptState("profile1");
28|       reelect;
29|     }
30|     asynchronous entry small (getAcceptState()=="profile1") {
31|       Thread.sleep(1000); /* delay= 1 second */
32|       batterylevel= batterylevel-10;
33|       if (batterylevel<80) setAcceptState("profile2");
34|       reelect;
35|     }
36|     asynchronous entry nothing (getAcceptState()=="profile2") {
37|       batterylevel= batterylevel-1;
38|     }
39|   }
40| }
```

Explication du code :
Ligne 3 : Trois entrées sont déclarées pour utiliser leurs informations dynamiques.
Lignes 9-22 : L'agent printer écrit dans le fichier de journalisation les informations sur les 3 sous-populations des agents impliquées dans le bloc group.
Lignes 24-29 : Le premier groupe effectue la tache la plus consommatrice en énergie. Pour cela les agents doivent être dotés de la perméabilité « `default` ». A noter qu'en absence d'instruction **setAcceptState**, c'est justement celle-ci qui est adoptée par les agents. Lorsque la batterie est insuffisamment chargée, la perméabilité courante devient « `profile1` ». Puis l'agent recommence éventuellement le traitement ou quitte le bloc
Lignes 30 à 35 : Le deuxième groupe effectue la tache un peu moins consommatrice en énergie. Pour cela les agents doivent être dotés de la perméabilité « `profile1` », un peu plus restrictive. Lorsque la batterie est insuffisamment chargée, la perméabilité courante devient « `profile2` ».

Lignes 36 à 40 : Le dernier groupe effectue la tache la moins consommatrice en énergie. Pour cela les agents doivent être dotés de la perméabilité « `profile2` », un peu plus restrictive. Puis l'agent quitte le groupe.

2.5.3 Des agents attentifs à des messages

```
01| import Default.xml as Default;
02| Default a1,a2,a3 = newAgent(Default);

03| synchronous entry main (true) {
04|   local int lID=.getProxyID();
05|   synchronous entry group (lID<2) {
06|     if (lID==0) {
07|       setAcceptState("onlyGetProxyIDbutUnrestricted");
08|       .log("["+lID+"] Waiting messages from my group or from others
agents\n");
09|       .log("["+lID+"] ");
10|       wait("onlyLogbutUnrestricted",2000);
11|       .log("["+lID+"] ");
12|       wait("onlyLogbutUnrestricted",2000);
13|       wait("nothing",1000);
14|       .log("["+lID+"] Waiting messages from my group only\n");
15|       .log("["+lID+"] ");
16|       wait("onlyLogRestrictedToUserAndGroup",2000);
17|       wait("nothing",1000);
18|     } else {
19|       .nop();
20|       .nop();
21|       .nop();
22|       logserver.log("Message from"+ lID +" as group\n");
23|       .nop();
24|       .nop();
25|       .nop();
26|       .nop();
27|       .nop();
28|       logserver.log("Message from"+ lID +" as group\n");
29|       .nop();
30|     }
31|   }
32|   synchronous entry other (lID==2) {
33|     .nop();
34|     .nop();
35|     .nop();
36|     .nop();
37|     .nop();
38|     logserver.log("Message from"+ lID +" as other\n");
39|   }
40| }
```

```
…
<Permeability permeability_name="nothing" umask="777"/>
<Permeability permeability_name="default" umask="000">
<Member member_id="ID_1" rights="111"/>
<Member member_id="ID_2" rights="111"/>
<Member member_id="ID_3" rights="111"/>
</Permeability>
<Permeability permeability_name="onlyGetProxyIDbutUnrestricted" umask="000">
<Member member_id="ID_1" rights="111"/>
<Member member_id="ID_3" rights="100"/>
```

```
</Permeability>
<Permeability permeability_name="onlyLogbutUnrestricted" umask="000">
<Member member_id="ID_3" rights="111"/>
</Permeability>
<Permeability permeability_name="onlyLogRestrictedToUserAndGroup" umask="000">
<Member member_id="ID_3" rights="110"/>
</Permeability>
<Permeability permeability_name="onlyLogRestrictedToUser" umask="000">
<Member member_id="ID_3" rights="100"/>
</Permeability>
</Agent>
```

Ligne 7 : Fixe pour le serveur de journalisation que toute invocation de méthode autre que `getProxyID` sera ignorée en tant qu'invocation externe. La méthode `log` pourra cependant être invoqué par lui-même.
Ligne 10 : Le serveur de journalisation devient attentif pour les invocations émanant de certains agents concernant certaines méthodes. Durant cette attention, il ne répond pas aux invocations pourtant conformes à sa perméabilité mais il ne les ignore pas : elles seront traitées dès que l'agent aura terminé son attention. Elle prend fin lorsqu'une méthode conforme est invoquée par un agent autorisé ou qu'un `timeout` apparaît.
Ligne 13 : Le serveur de journalisation attend jusqu'à ce qu'un `timeout` d'une seconde apparaît.
Ligne 22 : La demande de l'agent du groupe est synchrone avec l'attente du serveur. Ce synchronisme est obtenu par l'emploi des `nop()`.
Ligne 28 : La demande de l'agent du groupe est synchrone avec l'attente du serveur. Ce synchronisme est obtenu par l'emploi des `nop()`.
Ligne 38 : La demande de l'agent hors du groupe est synchrone avec l'attente du serveur. Ce synchronisme est obtenu par l'emploi des `nop()`.
Ligne 40 : l'agent hors du groupe attend les autres agents pour quitter le bloc `main`.

2.5.4 MASL et les agents réactifs à architecture subsumption et agents BDI

Pour pouvoir intégrer dans les missions des agents obéissant à l'architecture de la subsumption ou de l'architecture BDI, nous reprenons le framework GAA (Generic Agent Architecture) [Vidal & all, 02].
La classe de base de ce framework est `GAAActivity`. Un agent est défini en créant un certain nombre d'activités. Sa boucle de contrôle se charge alors de sélectionner celle à exécuter. Cette classe a trois méthodes principales : `canHandle`, `handle` et `inhibits`.
La méthode `canHandle` renvoie `true` si l'objet passé en paramètre peut être traité. En plus de son paramètre, elle peut aussi considérer l'état interne de l'agent et même sa représentation de l'environnement. Cette méthode précise les conditions pour que l'activité soit une solution appropriée.
La méthode `handle` est appelée lorsque l'activité a été choisie par la boucle de contrôle. Généralement, des primitives (actions atomiques) sont alors appelées. Des attributs de l'agent peuvent être modifiés. Le retour vaut `true` lorsque l'activité est terminée. Cependant elle peut être découpée en étapes lorsqu'elle implante des plans ou des comportements complexes de long terme. Les attributs servent alors à maintenir entre chaque phase un état. Une étape non terminale renvoie `false`.
La méthode `inhibits` reçoit en paramètre une `GAAActivity` et renvoie `true` si cette dernière est inhibée par l'activé courante. Ainsi cette méthode précise quand l'activité doit être exécutée.
Le framework est suffisamment souple pour permettre l'implantation d'une architecture de subsumption et des agents BDI ou une solution intermédiaire entre ces 2 solutions.
Pour un agent conforme à l'architecture de subsumption, une activité peut être inhibée par une autre. La méthode `inhibits` de la classe `GAAActivity` permet d'implanter cette possibilité. De plus les agents ne doivent pas maintenir des variables entre de multiples invocations d'une activité.
Un `BDIAgent` possède une collection de `GAADesire` qui représente ses désirs actifs. Ses activités doivent être construites de telle manière que leur méthode `canHandle` ne renvoie `true` que si son but fait partie des désirs actifs de l'agent. Dans la méthode `handle`, lorsque l'activité satisfait le désir, ce dernier est désactivé. Le framework n'impose pas une sémantique particulière à propos des intentions. Plusieurs systèmes BDI implantent différentes méthodes pour déterminer quant une intention doit être enlevée et quand une nouvelle intention est prioritaire. La méthode `inhibits` permet par exemple d'inhiber toutes les activités dont le but est moins prioritaire que celui de l'activité courante.

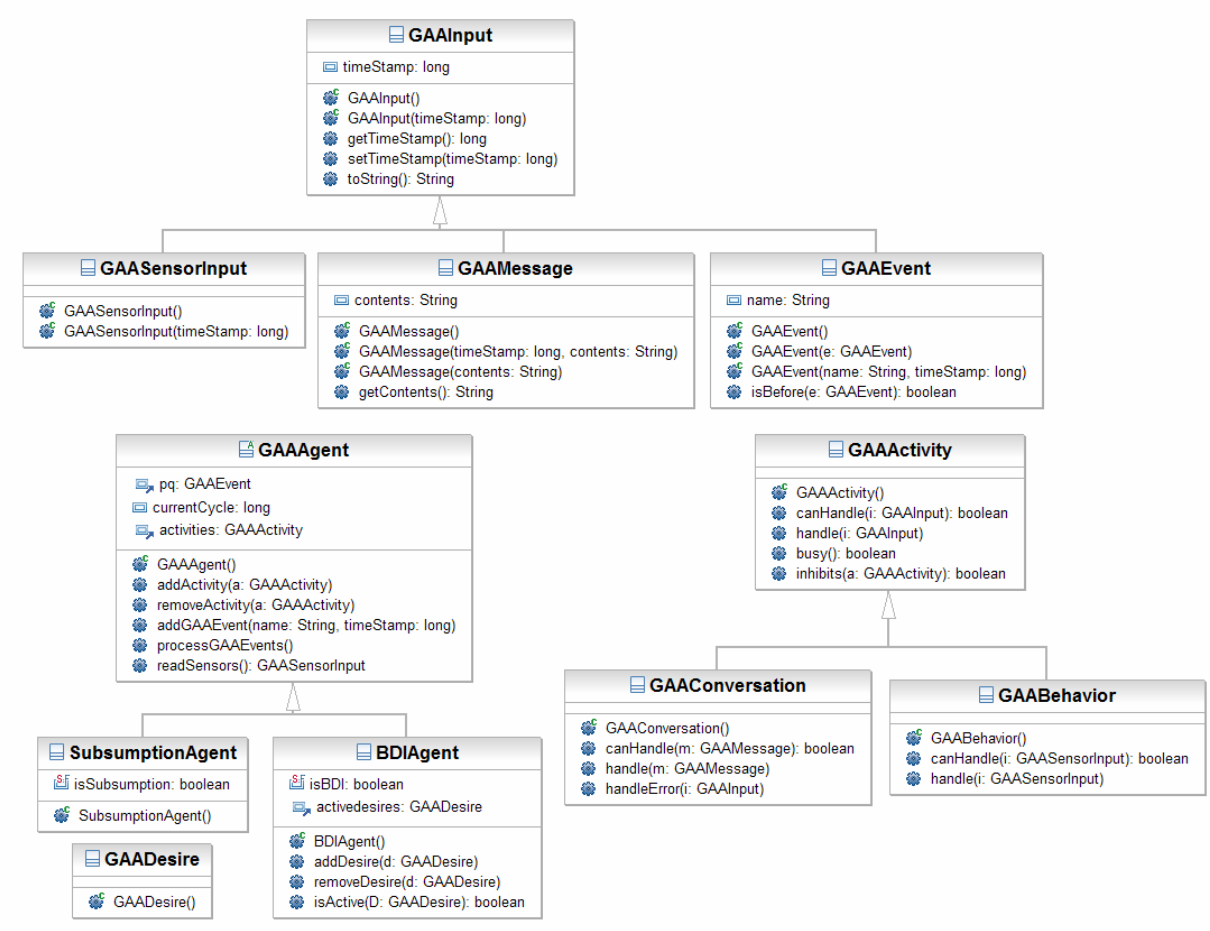


Figure 86 Les agents importés de GAA (Generic Agent Architecture) [Vidal & all, 02]

```
01| import Subsumption.xml as Subsumption;
02| import BDI.xml as BDI;

03| Subsumption a1,a2,a3 = newAgent(Subsumption);
04| BDI a4,a5,a6 = newAgent(BDI);

05| asynchronous entry main (true) {
06|   asynchronous entry GAA (.isSubsumption==true ||.isBDI==true ) {
07|     local Vector lmatches=new Vector();
08|     local Vector lunhibited=new Vector();
09|     local GAASensorInput lsensor;
10|     local Activity lchosen;
11|     local boolean linhibited=false;
12|     local int l;
13|     do {
14|       .processGAAEvents();
15|       lsensor=.readSensors();
16|       for (l=0; l<.activities.size(); l++)
17|         if (.activities.elementAt(l).canHandle(lsensor))
18|           lmatches.addElement(.activities.elementAt(l));
19|       for (l=0; l<lmatches.size(); l++) {
20|         linhibited=false;
21|         for (j=0; j<lmatches.size && l<>j; j++)
22|           if (lmatches.elementAt(j).inhibits(lmatches.elementAt(l)))
```

```
23|              linhibited=true;
24|          if (!linhibited)
25|            lunhibited.addElement(lmatches.elementAt(l));
26|        }
27|        lchosen=lunhinbited.elementAt(
new Random().nextInt(lunhibited.size()));
28|        if (lchosen.handle())
29|          .removeActivity(lchosen)
30|      } loop
31|    }
32| }
```

Explication du code :
La boucle de contrôle de l'agent reçoit les entrées des capteurs puis les assigne aux activités appropriées.
L'algorithme recherche toutes les activités qui peuvent traiter les entrées. De cet ensemble, il ne retient que celles qui ne sont pas inhibées par un autre membre. Puis il exécute une seule activité parmi ce sous ensemble.
Lignes 1,2 : les importations XML permettent de connaître les 2 types d'agents. A noter que leurs API définissent des objets passifs. Suite aux importations xml, les classes `Vector`, `Random`, `GAASensorInput`, etc. sont connues.
Lignes 14 : les évènements définis dans le framework GAA sont traités.
Lignes 7,8 : on déclare des variables puis on instancie des objets passifs par l'instruction new.
Ligne 15 : Les capteurs sont lus.
Lignes 16, 18 : Ne sont sélectionnées que les activités qui peuvent traiter les capteurs.
Lignes 19, 26 : Parmi les activités sélectionnées, on élimine celles qui sont inhibées par les autres.
Ligne 27 : L'activité à exécuter est choisie au hasard.
Lignes 28, 29 : Si l'activité est terminée, elle est enlevée des activités à exécuter.

Chapitre 3 : Le langage MASL

Dans ce chapitre nous donnons la syntaxe concrète de MASL et sa sémantique informelle. Après avoir donné la syntaxe BNF, nous aborderons le langage MASL en définissant l'ordre d'exécution des instructions MASL par un agent soumis à la nécessité de se synchroniser avec les autres membres de son groupe, qui doit réagir à des évènements et qui peut répondre aux sollicitations des autres agents. Les différents élements du langages que sont les types spécifiques (agent, entry, label et event), les variables MASL et les instructions classiques sont successivement abordés. Ce chapitre se veut synthétique et seuls des exemples éclairant des aspects non vus dans le chapitre 2 seront exposés.

3.1 BNF

MASL : :=
import file.xml [**as** agent_type_identifier]; *
agent_type_declaration; *
agents_set_declaration; *
[entry_bloc_declaration; *]

entry_mode **entry Main (true) {** MASL_declaration; *
MASL_instruction; *
[**react** (event) **{** react_instruction;***}**] *
[**react** (**default**) **{** react_instruction;***}**]
}

[A] agent_type_declaration ::= **type** agent_type_identifier **use** file.xml**;**

[B] agents_set_declaration ::=
agent_type_identifier identifier **= newAgent (**[agent_type_identifier]**);**

[C] entry_bloc_declaration ::= **Entry** entryname_list;

[D] entry_mode ::= **synchronous | synchronous | scalar**

[E] MASL_declaration::=
label_declaration [E1]
| local_variable [E2]
| shared_variable [E3]
| **local event** identifier_list; [E4]
| **shared event** identifier_list; [E5]

[E1] label_declaration::=
LLabel labelname_list
| **SLabel** labelname_list

[E2] local_variable ::=**local** type_indication identifier_list [= expression]

[E3] shared_variable ::=**shared** type_indication identifier_list
[**limited**(agents_type)] [= expression]

[F] MASL_instruction ::=
entry_member [F1]
| variable_assignment [F2]

| MASL_attribute_assignement [F3]
| MASL_method_call [F4]
| label_member [F5]
| **getAcceptState** : String [F6]
| **getAcceptState(**identifier**)** : String [F7]
| **setAcceptState(**String**)** [F8]
| **wait(**aSetOfMethodsFromXML : XMLTag); [F9]
| **wait(**aSetOfMethodsFromXML : XMLTag, timeout : integer); [F10]
| basic_instruction [F11]
| entry_bloc_definition [F12]
| **break** (level : integer) [F13]
| **break** (entryname) [F14]
| emit (e : event**)** [F15]
| **emitWithMessage(**e : event, message : String**)** [F16]
| **getMessageFromEvent(**e : event**)** : String [F17]
| **restart** [F18]
| **reelect(**level : integer**)** [F19]
| **reelect(**entryname**)** [F20]
| **clearEvents** [F21]

[F1] entry_member ::=
<entry_name>**.count**
| <entry_name>**.getWaitingInCount**() : integer
| <entry_name>**.getWaitingOutCount**() : integer
| <entry_name>**.getActiveCount():** integer
| <entry_name>**.getList()** : Object[]
| <entry_name>**.getWaitingInList**() : Object[]
| <entry_name>**.getWaitingOutList**() : Object[]
| <entry_name>**.getActiveList()** : Object[]
| <entry_name>**.lock()**
| <entry_name>**.release()**
| <entry_name>**.release(**count : integer**)**
| <entry_name>**.getLockPolicy()** : String

[F2] variable_assignment ::= identifier **=** expression

[F3] MASL_attribute_assignement ::=
.<identifier> **=** expression
| **this**.<identifier> **=** expression
| <agent name>.<identifier> **=** expression

[F4] MASL_method_call ::=
.method(parameters,*)
| **this**. method([parameters,*])
| <agent name>.method([parameters,*])

[F5] label_member ::=
<label_name>**.**<agent name>**.**method([parameters,*])
| <label_name>**.this.**method([parameters,*])
| <label_name>**.**<identifier> **=** <agent name>**.**method([parameters,*])
| <label_name>**.**<identifier> **= .this.**method([parameters,*])
| <label_name>**.**<agent name>.<identifier> **=** <agent name>**.**method([parameters,*])
| <label_name>**.**<agent name>.<identifier> **= .this.**method([parameters,*])
| <label_name>**.this.**<identifier> **=** <agent name>**.**method([parameters,*])
| <label_name>**.this.**<identifier> **= .this.**method([parameters,*])
| <label_name>**.isFinished()**: boolean
| <label_name>**.free()**

[F11] basic_instruction :==
MASL_if [G1]

MASL_loop	[G2]
| MASL_for	[G3]
| MASL_while	[G4]
| **exit (**level : integer**)**	[G5]

[G1] MASL_if ::= **if** (test) **{** MASL_instruction; ***}** **else {** MASL_instruction; ***}**

[G2] MASL_loop ::= **do {** MASL_instruction; ***} loop**

[G3] MASL_for ::= **for** (variable_assignement; test; expression)
{ MASL_instruction; ***}**

[G4] MASL_while ::= **while** (test) **{** MASL_instruction; ***}**

[F12] entry_bloc_definition ::=
entry_mode **entry** <entry name> (test) **{** MASL_declaration; *
MASL_instruction ;*
[**react** (event) **{** react_instruction; ***}**]*
[**react** (**default**) **{** react_instruction; ***}**]
}

[H] react_instruction ::=
MASL_instruction
| **resume**

Note that

type_indication ::= boolean | byte | char| short | int | float | long | double | utility_type_identifier

expression ::= **new** utility_type_identifier
| numeric_expression
| testing_expression
| logical_expression
| string_expression
| bit_expression
| **null**
| identifier
| MASL_method_call
| label_member
| entry_member

3.2 Langage cible des agents

Le langage MASL va être traduit en instructions dans le langage cible des agents, seul langage compréhensible par eux. De plus, la traduction s'appuie sur des appels à un runtime MASL dans le programme de l'agent.
Ce langage cible peut être c++ pour des Sbots par exemple. Il peut être aussi java par exemple pour un Khepéra.
Le processus de traduction MASL est exposé *Cf.* Annexe.

3.3 Séquencialité des instructions

L'exécution d'un programme de l'agent se fait instruction par instruction qui sont séparées par un « ; ». Le runtime MASL garantit qu'après l'exécution de chaque instruction du programme d'un agent, lorsque'un « ; MASL » est atteint, les traitements s'opèrent selon l'ordre Algorithme Séquencialité des Instructions ou algorithme S.D.I :

```
1 - tous les évènements perçus par l'agent sont traités (Cf.3.6) ;
2 - les invocations des primitives de l'agent par d'autres agents sont
effectuées. Cela ne veut pas forcément dire que le runtime attende que les
primitives non bloquantes terminent (Cf. 3.4.1) ;
3 - eventuellement, une synchronisation avec d'autres agents s'opère par
l'intermédiaire d'une barrière de synchronisation.
```

3.4 Typage

Le langage MASL définit des types obligatoires. Tout runtime MASL doit les implémenter. A un type, on fait correspondre une structure de données et des opérations permises sur le type.
Seront définis :
- les agents ;
- les blocs Entry ;
- les labels ;
- les évènements ;

3.4.1 Les agents

Le choix fait pour le langage MASL est celui d'une interface de classe statique (cf § 1.2.2). En effet, on considère que les capacités des robots sont connues a priori. Il sagit ici des capacités en terme de primitive d'action ou de perception du robot. Rien n'empêche que cette primitive soit programmée localement au sein du robot par une technique qui permette une certaine evolution de la primitive, mais du point de vue du service rendu, de nouvelles primitives ne peuvent être ajoutées dynamiquement. Le type de l'agent est définit dans un fichier XML. Nous faisons le choix d'une présentation systèmatique de ce dernier par opposition au chapitre 2 où la présentation se veut incrémentale et intuitive.

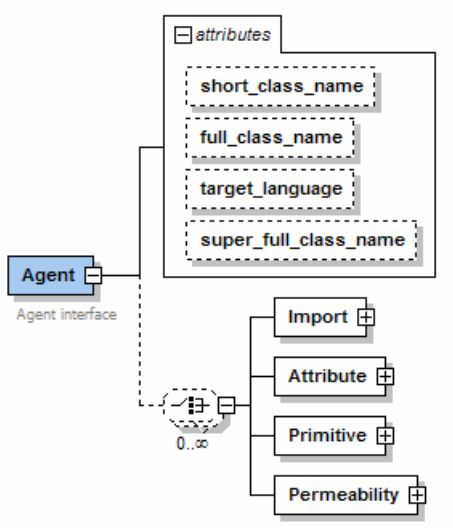
3.4.1.1 Importation et instanciation

Le type agent est défini par son API qui est déclarée dans un fichier XML. Sa prise en compte au niveau du runtime se fait par l'instruction import.

import file.xml [**as** agent_type_identifier]; *

Cette instruction met à disposition du runtime MASL la description du type d'agent, c'est à dire :
- des attributs ;
- des primitives ;
- de sa perméabilité.

Eventuellement est déclaré un type d'agent correspondant à ce fichier XML.



Le langage cible de l'agent (`target_language`) est utilisé par le moteur de traduction. La classe d'implémentation (`full_class_name`) et le nom par défaut du type (`short_class_name`) sont aussi définis. Eventuellement, une super classe est définie (`super_full_class_name`) afin de permettre l'instanciation d'un sous-type. Par contre, tous les membres de l'agent sont définis dans le fichier XML.

type agent_type_identifier **use** file.xml**;**

Cette instruction permet de définir le type contenu dans le fichier XML. Elle donne ainsi au programmeur la possibilité de ne pas connaître la valeur de `short_class_name`.
La description des attributs et des primitives contenues dans le fichier XML est faite dans des termes proches de la syntaxe java. Cependant, le runtime peut traduire cette description dans le langage hôte de l'agent.
A noter que cette instruction permet de différencier artificiellement les types pour pouvoir restreindre par exemple la capacité d'écriture sur des variables partagées (*Cf.* 3.5)

L'instanciation des agents

agent_type_identifier identifier **= newAgent (**[agent_type_identifier]**);**

Cette instruction permet d'instancier une collection d'agents de même type tout en leur attribuant des références explicites. L'instanciation est statique : elle ne peut s'opérer qu'en début de programme MASL.

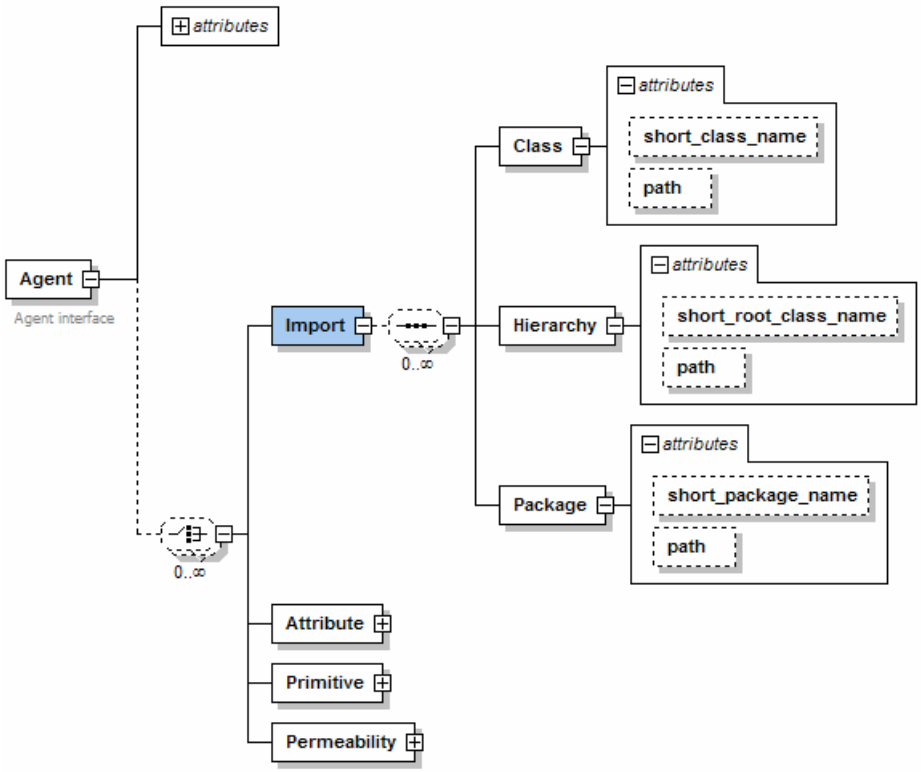
```
01| import Default.xml;
02| type Default use Default.xml;
03| type OtherDefault use Default.xml;
04| Default leader,a1,a2,a3 = newAgent(Default);
05| OtherDefault a4,a5 = newAgent(OtherDefault);
```

Dans cet exemple, les références explicites `leader`, `a1`, `a2`, `a3`, `a4`, `a5` seront accessibles dans tout le programme MASL. On remarque la distinction pour l'instant artificielle entre les types `Default` et `OtherDefault`.

Lors de l'instanciation, le runtime instancie la classe d'implémentation permettant d'accéder aux membres déclarés de l'API en appelant son constructeur sans paramètre puis définit l'ensemble des droits correspondant à la perméabilité (*Cf.* 3.4.1.5). Il en résulte une instance de l'agent ayant comme perméabilité courante, celle par défaut. Ainsi le runtime sera en mesure de conditionner l'accès à tous les membres des agents à la perméabilité courante de l'instance d'agent. L'instanciation peut utiliser un sous-type du type déclaré.

L'instanciation met à disposition de la mission un agent ainsi que les membres de cet agent qui correspondent à l'ensemble de ses attributs et de ses méthodes.

3.4.1.2 Les importations d'objets passifs d'un agent



L'API d'un agent est centrée sur un objet actif qui possède son propre flux de contrôle. Cette classe peut utiliser des objets passifs. Tous ces objets partagent le même langage cible. Les objets passifs sont implicitement importés avec l'agent. En effet si un attribut (*Cf.* 3.4.1.3) ou un paramètre de méthode (*Cf.* 3.4.1.4) ont un type non élémentaire, la manipulation de ces types est d'intérêt pour un programme MASL. L'importation vaut aussi pour les variables partagées ou locales (*Cf. 3.5*).
Plusieurs facilités d'importations de classes utilitaires sont possibles :

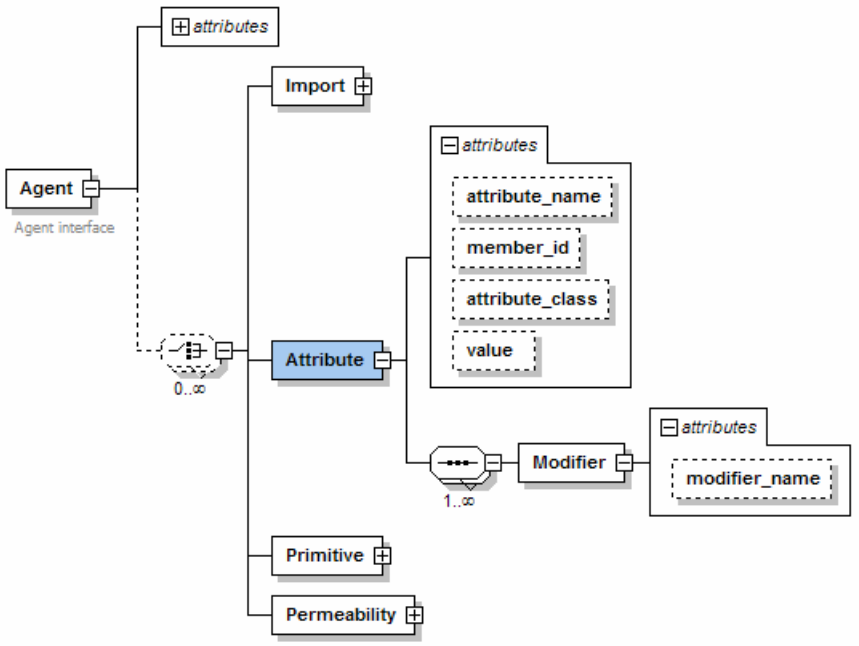
- Importation d'une classe précise par l'élément XML `Class` ;
- Importation d'une hiérarchie de classes en précisant la classe racine de l'héritage ;
- Importation de toutes les classes d'un paquetage.

Ces possibilités dépendent du langage cible. De plus la signification de `path` en est fonction.

expression ::= **new** utility_type_identifier
| …

Cette instruction permet d'instancier une classe utilitaire (objet passif) qui est définie dans l'API de l'agent.

3.4.1.3 Les attributs d'un agent



Même si on a la possibilité d'implémenter par des méthodes `getX`/`setX` l'accès aux attributs, l'API peut aussi donner la possibilité d'y accéder directement. Elle est prise en compte dans le fichier XML. L'`attribute_name` et l'`attribute_class` définissent le nom de l'attribut ainsi que son type. Les types correspondent aux types élémentaires des langages cibles (*Cf.* 3.2) ou des tableaux de ces types quand ils sont supportés ou des classes si c'est possible. Des fichiers de description de ces classes sont alors disponibles. On les appelle des classes utilitaires. Ils entrent dans la catégorie des objets passifs alors que les agents sont des objets actifs. Les types élémentaires autorisés pour les attributs de l'API correspondent aux types élémentaires des variables (*Cf. 3.5*).
L'utilisation des attributs comprend l'affectation d'autres attributs, affectation/initialisation de variables locales ou partagées, paramétrage lors de l'invocation de méthodes, évaluation lors d'un test de E-bloc ou de celui d'un **if** ou d'une condition de poursuite d'une boucle.
Les attributs concernent les variables d'instances, les variables de classes, les constantes qui sont publiques. Ce sont les modificateurs prévus dans la description XML qui permettent de faire ces distinctions.
Le modificateur `static` définit une variable de classe : elle est présente en un seul exemplaire dans la classe et elle est partagée par toutes les instances.
Le modificateur `final` définit une constante : elle prend sa valeur seulement à l'initialisation. On ne peut plus la changer par la suite.

```
<Attribute attribute_name ="variable_instance" member_id ="ID_0" attribute_class ="int.class">
</Attribute>
<Attribute attribute_name ="variable_classe" member_id ="ID_1" attribute_class ="int.class">
   <Modifier modifier_name ="static"/>
</Attribute>
<Attribute attribute_name ="constante" member_id ="ID_2" attribute_class ="int.class" value="180">
   <Modifier modifier_name ="static"/>
   <Modifier modifier_name ="final"/>
</Attribute>
< Attribute attribute_name ="type_classe" member_id ="ID_3" attribute_class ="masl.khepera.Vector2d">
</Attribute>
```

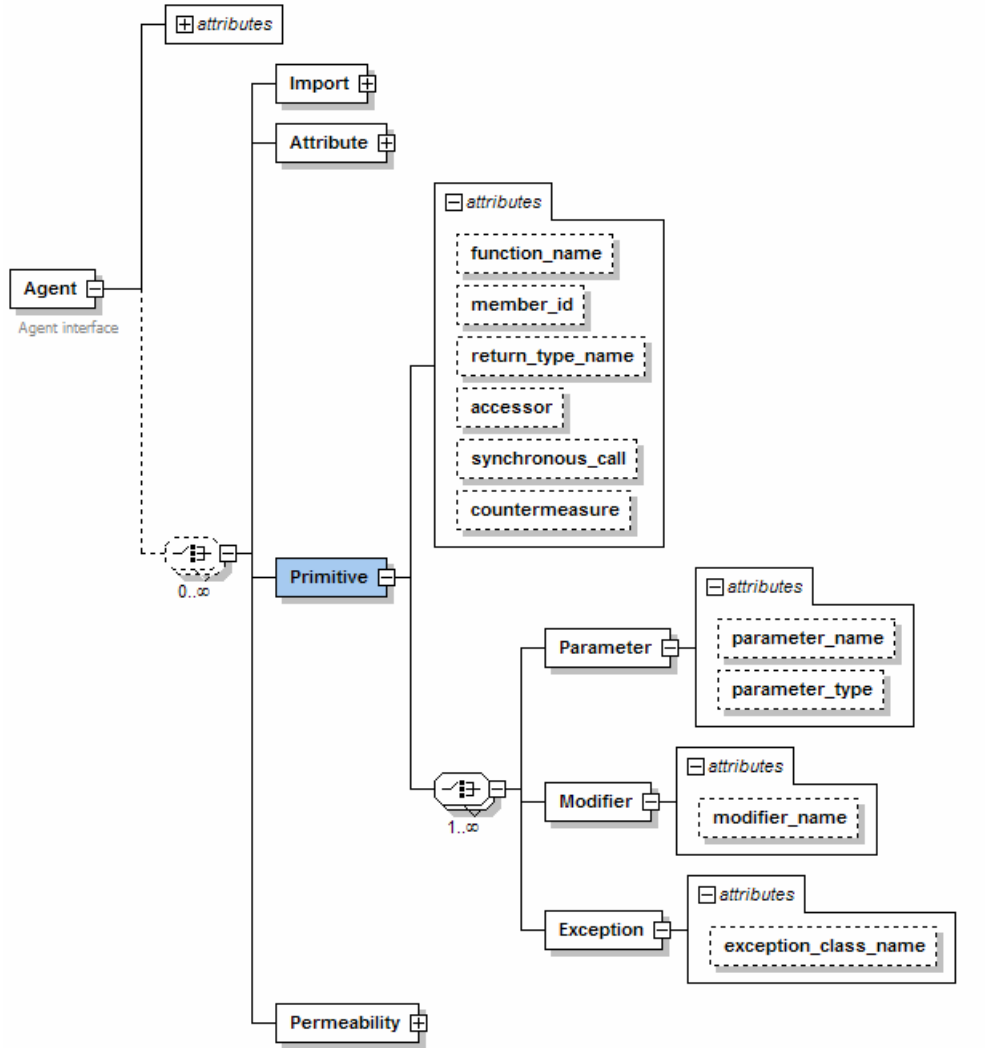
[F3] MASL_attribute_assignement ::= .<identifier> **=** expression
| **this**.<identifier> **=** expression
| <agent name>.<identifier> **=** expression

En absence de référence explicite (<agent name>**.**variable), ou lorsque <agent name> correspond à l'agent courant, on est en présence des variables d'instance ou de classe de l'agent courant. La notion d'agent courant est liée à la notion de E-bloc (*Cf.* 3.4.2) et fait référence à tous agent membre du E-bloc qui exécute l'instruction utilisant l'attribut. A noter que syntaxiquement, l'accès à une variable d'instance ou de classe n'est pas différencié, contrairement au langage java.
Par rapport à l'algorithme SDI (*Cf. 3.1*), la lecture et l'affectation des variables d'instance et de classe sont immédiates, en tant qu'instruction courante.
Avec cette définition, un programme d'un agent peut comprendre :

```
06| {Mode} entry Main (true) {
07| a4.variable_instance=4;
08| a4.variable_instance=.variable_instance;
09| a5.variable_instance=.constante;
10| a6.variable_instance=.variable_classe;
11| .variable_instance=5;
12| .variable_instance=a6.variable_classe;
13| .variable_classe =a6.variable_instance;
14| .variable_instance =.variable_classe;
15| .type_classe= new masl.khepera.Vector2d(0,0);
16| .type_classe= new Vector2d(0,0);
17| }
```

Dans cet exemple, la variable d'instance `variable_instance` de l'agent `a4` se voit affecter d'abord la valeur 4 puis cette prise par l'attribut de même nom de l'agent courant. La variable d'instance `variable_instance` de l'agent `a5` se voit affecter la valeur correspondant à l'attribut constant de l'instance courante. La variable d'instance `variable_instance` de l'agent `a6` prend la valeur de la variable de classe `variable_classe`. Pour l'agent courant, sa variable d'instance et de classe peuvent être affectées par des valeurs provenant de lui-même ou d'autres instances référencées explicitement.
Ligne 15, la variable `type_classe` reçoit une instance d'une classe utilitaire.
Ligne 16, on utilise l'importation implicite des classes utilitaires pour obtenir le même résultat que la ligne précédente.

3.4.1.4 Les primitives d'un agent



[F4] MASL_method_call ::=
.method(parameters,*)
| **this**. method([parameters,*])
| <agent name>.method([parameters,*])

L'invocation de primitives distingue syntaxiquement l'invocation interne et externe. L'invocation interne est l'instruction courante du programme de l'agent. L'invocation externe émane d'un autre agent. Elle n'est exécutée qu'à la terminaison de l'instruction courante du programme de l'agent destinataire (*Cf.* 3.3). Lorsque la référence explicite de l'agent correspond à l'agent lui-même, il s'agit alors d'une invocation interne.

Un agent dispose de primitives. Tout d'abord une primitive comprend un type retour et des paramètres. Les types de ces paramètres et retour correspondent aux types élémentaires des langages cibles (*Cf.* 3.2) ou des tableaux de ces types quand ils sont supportés ou des classes si c'est possible. Des fichiers de description de ces classes sont alors disponibles.

Puis la description comporte :

- le mode d'invocation (bloquant/non bloquant) ;
- la méthode de compensation ;
- le modificateur ;
- les exceptions levées par la primitive.

Une primitive est généralement invoquée de manière synchrone (appel bloquant). L'agent ne reprend son contrôle qu'après la terminaison de la primitive. Si elle est invoquée de manière asynchrone, l'agent a la possibilité de faire d'autres tâches. Il sera notifié de la terminaison de la méthode. En cas d'utilisation du retour d'une invocation asynchrone d'une fonction, l'agent est bloqué jusqu'à ce que le retour soit disponible.

Dans le cas d'une invocation asynchrone, l'API de l'agent peut préciser une méthode de compensation : méthode qui doit être invoquée pour remettre l'agent dans un état compatible par rapport à son bon fonctionnement en cas de changement de mission. La conception de l'API veut qu'en cas d'appel bloquant, la primitive lors de son retour met l'agent automatiquement dans un état cohérent. En cas d'appel asynchrone à une primitive qui prend un certain temps, la remise en cohérence du fait du retour peut ne pas intervenir, puisque que l'invocation peut être interrompue. Dans ce cas, elle peut être forcée par l'appel à une primitive adhoc. Elle est automatiquement gérée par le runtime.
Les invocations internes sont exécutées immédiatement en tant qu'instruction courante puisque elles font partie du programme de l'agent. Conformément à l'algorithme S.D.I. (cf. 3.3), les invocations externes sont généralement différées. L'instruction « **wait** » permet de rendre immédiate une invocation externe. Elle demande à l'agent d'être attentif à une invocation externe qui sera considérée comme l'instruction courante. Si l'invocation externe n'intervient pas avant un certain délai précisé, l'instruction **wait** termine. Sans cette instruction, ce n'est qu'après l'exécution de l'instruction courante de l'agent que les invocations externes émanant des autres agents sont prises en compte.
L'`accessor` permet de déclarer qu'une primitive qui renvoie un résultat doit être considérée par le runtime MASL/processus de traduction comme ayant une durée nulle, c'est-à-dire qu'elle n'est pas sujette à une barrière de synchronisaton (*Cf.* 3.4.2).
Le modificateur `static` précise que l'on est en présence d'une méthode de classe : cette méthode ne modifie pas l'état des instances. Seules les variables de classes sont accessibles en écriture.
Une exception non capturée est promue en évènement local.

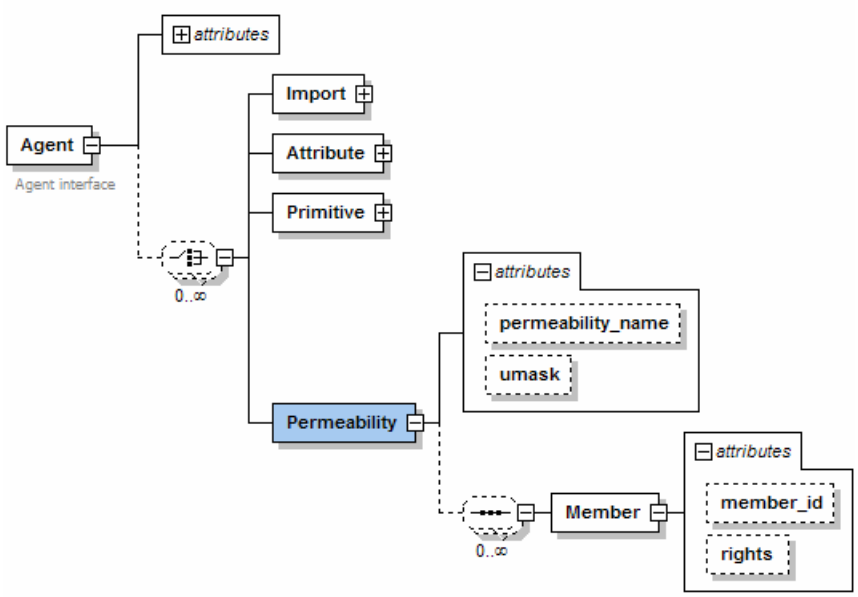
```
  <Primitive function_name="getProxyID" member_id="ID_3" synchronous_call="false" return_type_name="
int.class" accessor=" true "  >
 </Primitive>
  <Primitive function_name ="MoveBackward" member_id ="ID_3" synchronous_call ="true"
return_type_name ="void">
    <Parameter parameter_name ="distance" parameter_type="int.class"/>
  </Primitive>
  <Primitive function_name="setfSomethingVeryDangerousOn" member_id="ID_7" synchronous_call="false"
return_type_name=" void" countermeasure=" setfSomethingVeryDangerousOff "  >
 </Primitive>
  <Primitive function_name="goto_point" member_id="ID_8" synchronous_call="true"
return_type_name="masl.khepera.Trajectory" >
   <Modifier modifier_name ="static"/>
   <Parameter parameter_name ="position" parameter_type=" masl.khepera.Vector2d"/>
   <Parameter parameter_name ="velocity" parameter_type=" masl.khepera.Vector2d"/>
   <Parameter parameter_name ="angle" parameter_type=" double.class "/>
   <Parameter parameter_name ="target_p" parameter_type=" masl.khepera.Vector2d"/>
   <Parameter parameter_name ="target_v" parameter_type=" masl.khepera.Vector2d"/>
   <Parameter parameter_name ="target_a" parameter_type=" double.class "/>
 </Primitive>
```

Avec cette définition, un programme d'un agent peut comprendre :

```
06| {Mode} entry Main (true) {
07| a4.MoveBackward(10);
08| .MoveBackward(10);
09| .goto_point(new masl.khepera.Vector2d(0,0), new masl.khepera.Vector2d(
     1,0),0.36, new  masl.khepera.Vector2d(0,0), new masl.khepera.Vector2d(
     1,0),0.36);
10| .setfSomethingVeryDangerousOn();
11| }
```

Dans cet exemple, l'agent courant demande à l'instance référencée explicitement par a4 d'avancer de 10.Puis, il recule de 10. Ensuite il appelle sa méthode de classe qui calcule une trajectoire en prenant en compte les données issue d'une caméra au dessus du terrain pour éviter des obstacles. Enfin, il procède à une action qui provoquera explicitement une action compensatoire avant la fin de son programme.

3.4.1.5 La perméabilité d'un agent



La perméabilité est un mécanisme fortement inspiré dans sa forme par les droits unix de la commande `chmod` qui permet de définir des droits d'utilisation des attributs et des primitives.qui gèrent l'autonomie des agents en fonction de la provenance de la demande d'accès. Ainsi un agent nommé « appelant » peut demander à un autre agent « appelé » qu'il exécute une primitive ou qu'il modifie ses attributs. On définit aussi des états de perméabilités comme la définition statiques des droits sur des membres. Elle permet la définition unifiée et dynamique des droits d'accès en lecture (`r`), en écriture (`w`) pour les attributs et en exécution (`x`) pour les primitives de l'API. L'accès en lecture et/ou en écriture est immédiat s'il est autorisé : il se fait en tant qu'instruction courante de l'agent demandeur. Le refus en lecture entraîne l'émission d'un évènement local dans l'agent demandeur (*Cf. 3.6*).

La perméabilité définit s'il ignore une invocation : la méthode peut ne pas être invoquée pour soi-même, pour un agent co-équipier (qui joue le même rôle=présent dans le même bloc entry *cf. 3.4.2*) ou pour un autre agent quelconque. A la manière des droits unix, on définit les droits utilisateur (`u`) qui concernent l'accès interne aux membres de l'instance, les droits groupe (g) qui régissent l'accès aux membres par des agents eux aussi en cours d'exécution dans le même bloc entry et les droits des autres agents (`o`).

Le fichier XML contient un catalogue des états de perméabilité qui représentent toutes les évolutions possibles au cours d'une mission.

La perméabilité, lorsqu'elle définit les accès à l'API pour l'agent lui-même, permet de définir les modes dégradés. Ainsi le fichier XML publie une liste d'états possibles que l'on peut constater ou déclencher par les instructions **getAcceptState** et **setAcceptState**.

| **getAcceptState** : String
| **getAcceptState(**identifier**)** : String
| **setAcceptState(**String**)**
| **wait(**aSetOfMethodsFromXML : XMLTag);
| **wait(**aSetOfMethodsFromXML : XMLTag, timeout : integer);

L'instruction **getAcceptState** donne la perméabilité de l'agent voire celle d'un autre agent.
L'instruction **setAcceptState** fixe la perméabilité courante de l'agent.
L'instruction « **wait** » permet de rendre immédiate une invocation externe. Elle demande à l'agent d'être attentif à une invocation externe qui sera considérée comme l'instruction courante. Le premier paramètre définit l'ensemble des appels externes autorisés. Seules les méthodes explicitement autorisées en exécution pour l'appelant dans l'état de perméabilité précisé seront possibles. Le premier appel externe conforme sera l'instruction courante. Si l'invocation externe n'intervient pas avant un certain délai précisé, l'instruction **wait** termine. Sans cette instruction, ce n'est qu'après l'exécution de l'instruction courante de l'agent que les invocations externes émanant des autres agents sont prises en compte (*Cf. 3.3*).

La perméabilité, lorsqu'elle définit les accès à l'API pour les autres agents peut être l'extension des modes dégradés (Pourquoi servir les autres alors que l'on ne peut plus se servir soi-même ?), ou peut être l'autonomie de l'agent. La coordination d'une équipe peut nécessiter que le niveau d'autonomie d'un agent soit fonction de l'agent lui-même mais aussi de son équipe et en particulier des autres membres. Par exemple, le rejet d'un service peut être fait pour des considérations locales à l'agent (pas assez d'énergie). La motivation du refus peut être aussi par exemple la présence de coéquipiers suffisamment nombreux ou mieux dotés que l'agent.

Pour les accès aux variables d'instances et aux méthodes d'instance, les accès sont filtrés en fonction de la seule perméabilité courante. Pour les accès aux variables de classes et aux méthodes de classes qui offrent des services partagés par toutes les instances, les agents qui partagent le même type ne seront limités que s'ils se limitent eux-mêmes par une exclusion des droits user puisque chaque agent a autant de légitimité sur les accès à sa classe. Par contre les agents ayant un type différent voient leur accès aux membres de la classe de l'autre agent filtrés uniquement par la perméabilité courante de l'agent destinataire du message.

3.4.2 Le type Entry

L'intégration est possible grâce à la construction `entry` qui est inspirée de la construction `switch` / `break` de java. Dans ce langage, le calcul de la mission est exprimé par un seul programme qui sera déployé sur une plateforme SMA.
La programmation de la mission de l'équipe recouvre trois aspects : le rythme, le temps et l'espace.
Le rythme est marqué par l'exécution séquentielle, le synchronisme, l'asynchronisme et aussi par les changements de synchronisme au cours du temps.
Lorsqu'ils se sont en séquence, les blocs entry expriment la mission du groupe dans le temps.
Lorsqu'ils sont emboités, la programmation est exprimée selon l'espace : les agents satisfont la condition locale d'entrée dans le bloc en fonction de leur état interne mais aussi de leur perception de l'environnement qui varie avec la position de l'agent dans le monde.

Concept de E-bloc
Un E-bloc, c'est une séquence d'instructions à exécuter conditionnée à l'évaluation favorable d'un test. Un agent qui se détermine favorablement par rapport au test, entre dans le E-bloc : il devient membre de ce E-bloc et il exécute séquentiellement ses instructions. Dans un E-bloc, des variables peuvent êtres définies, initialisées et utilisées (*Cf.* 3.5). De plus, le E-bloc précise la réaction de ces membres à la réception d'évènements. Il y a trois modes d'exécution de E-bloc : le mode synchrone, le mode asynchrone et le mode scalaire. Le mode asynchrone permet l'exécution parallèle des agents. Le mode synchrone oblige la synchronisation des agents après chaque instruction. Le mode scalaire exclut le parallélisme : seul un agent parmi ceux élus exécutera le bloc.
Les E-blocs peuvent être imbriqués : le E-bloc contenant le bloc courant sera désigné comme E-bloc père. Le E-bloc contenu dans un bloc sera désigné comme E-bloc fils.
Un agent ayant fini l'exécution de la dernière instruction qu'il a à exécuter du bloc `entry` le quitte de manière normale. L'instruction break le fait partir de manière prématurée. Il en est de même suite à la réaction aux évènements.

```
[F12] entry_bloc_definition ::=
   entry_mode entry <entry name> (test) { MASL_declaration; *
         MASL_instruction; *
         [react (event) { react_instruction;*} ] *
         [react (default)          { react_instruction;*} ]
   }
   [D]  entry_mode ::= synchronous | synchronous | scalar
```

Concept de E-Sequence
Lorsque pour un agent, l'instruction à exécuter est un E-bloc, il doit s'évaluer par rapport au test. Si l'évaluation est positive, l'agent devient un membre du bloc. Si ce n'est pas le cas, l'instruction à exécuter est l'instruction après le bloc puisque le test est de durée nulle. Cette dernière peut être un autre E-bloc. Dans ce cas l'agent doit encore s'évaluer par rapport au nouveau test et ainsi de suite. On appelle E-Séquence, une séquence ininterrompue de E-bloc (qui n'est pas réduite à un seul E-bloc). L'exécution des autres E-bloc suivant de la E-Séquence se fera de manière conditionnelle, ce qui diffère de la construction `switch` / `break` de java.

```
yet executed                | .moveForward(10);
Start of the E-sequence     |
Test of the first E-bloc    | synchronous entry e1 (.isTE1()){
To be executed now?         |   .moveLeft(30);
                            |  …
End of the first E-block    | }
Test of the second E-bloc   | asynchronous entry e2 (.isTE2()){
To be executed now?         |   .moveRight(30);
                            |  …
End of the second E-block   | }
End of the E-sequence       |
To be executed now?         | .moveForward(10);
```

Figure 87 Une E-Sequence et son évaluation par les agents.

E-bloc asynchrone

Un E-bloc asynchrone est une simple exécution parallèle de ses agents membres sans aucun mécanisme de synchronisation. Les agents qui exécutent les instructions du E-bloc de manière séquentiel sont considérés comme des membres actifs.

E-bloc scalaire

Un E-bloc scalaire est une exécution séquentielle restreinte à un seul de ses agents membres. Ce type de E-bloc se caractérise par le fait qu'il sera exécuté par au maximum un agent. En effet, parmi tous les agents qui se sont déterminés favorablement au test du E-bloc, un seul est choisi arbitrairement pour l'exécuter. Ceux qui n'ont pas été choisi arbitrairement sont considérés comme l'ayant exécuté.

E-bloc synchrone

Un E-bloc synchrone est une exécution parallèle synchronisée de ses agents membres. Il se caractérise par une barrière de synchronisation, par une barrière d'entrée synchronisée et par une barrière de sortie synchronisée.

- Barrière de synchronisation

La barrière de synchronisation ne concerne que les membres actifs du E-bloc et ne s'applique qu'après l'exécution d'une instruction de durée non nulle. Elle consiste à vérifier que tous les agents actifs pour ce bloc ont terminé l'exécution de leur instruction courante. Tant que ce n'est pas le cas, les agents attendent. Si c'est les cas, la barrière les libère. Ils peuvent alors exécuter les instructions suivantes.
Toutes les instructions sont considérées à durée non nulle sauf : les méthodes des E-bloc, les méthodes des Labels (*Cf. 3.4.3*), les méthodes accesseurs de l'API des agents (*Cf. 3.4.1)*.

- Barrière d'entrée synchronisée

La barrière d'entrée synchronisée consiste à bloquer tout agent qui vient de se déterminer favorablement au test du E-bloc. Il est déjà considéré comme un agent membre. Mais il ne peut pas exécuter la première instruction. Les agents sont considérés comme étant en attente d'entrée.
Lorsque le E-bloc n'a jamais eu de membre, elle ne libérera les agents en attente que lorsque tous les agents qui doivent se positionner dans la E-séquence ont tous déterminé leur prochaine instruction. La barrière est considérée comme libre au sens que la libération des agents est automatique. Une fois qu'elle a libérée pour la première fois des agents, elle devient fermée : seul des demandes explicites **.release**() et **.release**(count) peuvent libérer les agents en attentes.

- Barrière de sortie synchronisée

Dans un bloc synchrone, lorsqu'un agent quitte normalement le bloc, c'est-à-dire lorsqu'il a exécuté la dernière instruction qu'il peut exécuter du bloc, il est mis en attente de sortie. Les agents ne quittent cette barrière que lorsque le bloc est vide : il n'y a plus d'agents actifs. A noter que les agents qui exécutent une instruction **break** ou qui quittent le bloc suite à un **react** ne sont pas soumis à cette barrière.

Concept de vagues synchrones d'agents

Plusieurs membres d'un bloc peuvent exécuter entre chaque barrière de synchronisation la même instruction. Il s'agit d'une exécution redondante qui est généralement recherchée. Plusieurs moyens concourent déjà pour l'obtenir : la barrière d'entrée et la barrière de synchronisation éventuellement partagée avec un bloc père. Une première instruction **.release**() conditionnée dans un **if** faisant appel à des accesseurs dans la condition permet aussi d'intégrer des nouveaux éléments à la vague car elle est considérée à durée nulle.

```
01| import Default.xml as Default;
02| Default a0, a1, a2 = newAgent(Default);
03| Entry e1;

04| …
05|   synchronous entry e1 (.getProxyID()%2) {
06|     do {
07|       if (.getProxyID()==0) e1.release();
08|       .log("Hello");
09|       .log(" world !");
10|     } loop
11|   }
```

Sortie d'E-bloc

Un agent est membre d'un E-bloc tant qu'il a une instruction dans ce E-bloc à exécuter. Un agent qui a exécuté sa dernière instruction le quitte. Il le quitte aussi s'il ne satisfait pas le test concerné par les dernières instructions.

| **break** (level : integer) [F13]
| **break** (entryname) [F14]

L'instruction **break** fait quitter le E-bloc concerné : la prochaine instruction à exécuter est l'instruction juste après sa fin. De plus une réaction à un évènement peut être aussi l'origine du départ de l'agent du bloc. Ces cas sont des sorties anticipées d'un E-bloc et peuvent donner lieu à des appels automatiques à des primitves de compensation.

Déclaration des E-bloc

[C] entry_bloc_declaration ::= **Entry** entryname_list;

La déclaration des E-bloc est un préalable à l'utilisation des méthodes attachées à cette structure. Tout programme MASL possède un E-bloc main. Sa déclaration est implicite.

Utilisation des informations du type entry

Suite à la déclaration du bloc, les méthodes suivantes peuvent être utilisées :

[F1] entry_member ::=
<entry_name>**.count**
| <entry_name>**.getWaitingInCount**() : integer
| <entry_name>**.getWaitingOutCount**() : integer
| <entry_name>**.getActiveCount():** integer
| <entry_name>**.getList()** : Object[]
| <entry_name>**.getWaitingInList**() : Object[]
| <entry_name>**.getWaitingOutList**() : Object[]
| <entry_name>**.getActiveList()** : Object[]
| <entry_name>**.lock()**
| <entry_name>**.release()**
| <entry_name>**.release(**count : integer**)**
| <entry_name>**.getLockPolicy()** : String

Tableau 3 Le résultats des méthodes d'un bloc Entry

	Résultat des méthodes
E-Bloc asynchrone	**.count** / **.getList()** renvoie le nombre ou la liste des agents membres du bloc. **.getWaitingInCount**() / **.getWaitingInList**() renvoie 0 / une liste vide. **.getActiveCount()** / **.getActiveList()** renvoie les mêmes résultats que **.count** / **.getList()**. **.getWaitingOutCount**() / **.getWaitingOutList**() renvoie 0 / une liste vide. **.getLockPolicy()** renvoie la valeur « unavailable ». **.release()** / **.release(**count : integer**)** n'ont aucun effet.
E-Bloc synchrone	**.count** / **.getList()** renvoie le nombre ou la liste des agents membres du bloc. **.getWaitingInCount**() / **.getWaitingInList**() renvoie le nombre ou la liste d'agents membres mis en attente par la barrière d'entrée. **.getActiveCount()** / **.getActiveList()** renvoie le nombre ou la liste d'agents membres qui doivent encore exécuter une instruction du bloc **.getWaitingOutCount**() / **.getWaitingOutList**() renvoie le nombre ou la liste d'agents membres mis en attente par la barrière de sortie.

	.getLockPolicy() renvoie la valeur « free » si la première vague d'agent est attendue. Elle renvoie « locked » si la première vague d'agent est déjà entrée ou si on a appliqué **.lock()**. **.release()** / **.release(**count : integer**)** libère tous les agents ou une partie seulement des agents de la barrière d'entrée
E-Bloc scalaire	**.count** / **.getList()** renvoie le nombre ou la liste des agents membres du bloc. **.getWaitingInCount**() / **.getWaitingInList**() renvoie 0 / une liste vide. **.getActiveCount()** / **.getActiveList()** renvoie 1 au maximum et la liste ne peut comporter qu'un agent : celui qui doit encore exécuter une instruction du bloc **.getWaitingOutCount**() / **.getWaitingOutList**() renvoie 0 / une liste vide. **.getLockPolicy()** renvoie la valeur « unavailable ». **.release()** / **.release(**count : integer**)** n'ont aucun effet.

Voici un exemple qui illustre cette utilisation.

```
01| import Default.xml as Default;
02| Default a0, a1, a2 = newAgent(Default);
03| Entry nothing;

04| asynchronous entry main (true) {
05|   asynchronous entry nothing (.getProxyID()%2) {
06|     .log(main.count);    //entry définie par défaut
07|     .log(nothing.count); //entry définie ligne 3
08|   }
09| }
```

Imbrication des E-bloc

Les E-blocs peuvent être imbriqués afin d'obtenir un rythme précis entre agents. Ce dernier dépend en effet du type du bloc père et celui du bloc père. La racine de l'imbrication est le bloc `main`.

Lorsque le E-bloc fils est scalaire, sa réexécution est conditionnée au fait que le E-bloc père se soit vidé entre temps.

Tous les agents membres du bloc fils sont considérés par le bloc père dans **.getActiveCount()** / **.getActiveList()** et donc dans **.count** / **.getList()**.

```
01| import Default.xml as Default;
02| Default a0, a1, a2 = newAgent(Default);

03| {ModePere} entry pere (true) {
04|   .log("Hello ");
05|   .log("world !");
06|   {ModeFils} entry fils (true) {
07|     .log("Hello ");
08|     .log("world !");
09|   }
10| }
```

Voici la matrice d'imbrication :

Tableau 4 Matrice d'imbrication d'un bloc entry

Imbrications de blocs	***{ModePere}*=asynchronous**	***{ModePere}*=synchronous**	***{ModePere}*=scalar**
***{ModeFils}*= Asynchronous**	Aucune barrière de synchronisation à chaque instruction. Aucune barrière d'entrée synchronisée. Aucune barrière de sortie synchronisée. La différence de rythmes et de réaction aux évènements justifie cette imbrication	Barriere de synchronisation à chaque instruction dans le bloc fils inactive. Barrière d'entrée synchronisée dans le bloc fils inactive. Barrière de sortie synchronisée dans le bloc fils inactive. Tout le bloc est considéré comme une instruction par le bloc père qui lui applique en fin sa barrière de synchronisation. La différence de rythmes et de réaction aux évènements justifie cette imbrication	Imbrication interdite car sans objet.

Imbrications de blocs	***{ModePere}*=asynchronous**	***{ModePere}*=synchronous**	***{ModePere}*=scalar**
***{ModeFils}*= Synchronous**	Barriere de synchronisation à chaque instruction est active dans ce bloc Barrière d'entrée synchronisée dans le bloc fils active. Barrière de sortie synchronisée dans le bloc fils active La différence de rythme et de réaction aux évènements justifie cette imbrication	Barriere de synchronisation à chaque instruction identique entre le bloc père et bloc fils Barrière d'entrée synchronisée dans le bloc fils active. Barrière de sortie synchronisée dans le bloc fils active La différence de réaction aux évènements justifie cette imbrication	Imbrication interdite car sans objet.

Imbrications de blocs	***{ModePere}*=asynchronous**	***{ModePere}*=synchronous**	***{ModePere}*=scalar**
***{ModeFils}*= Scalar**	Barriere de synchronisation à sans objet du fait de l'absence de parallélisme Barrière d'entrée synchronisée sans objet du fait de l'absence de parallélisme	Barriere de synchronisation à sans objet du fait de l'absence de parallélisme Barrière d'entrée synchronisée sans objet du fait de l'absence de parallélisme	Barriere de synchronisation à sans objet du fait de l'absence de parallélisme Barrière d'entrée synchronisée sans objet du fait de l'absence de

	Barrière de sortie synchronisée sans objet du fait de l'absence de parallélisme. La différence de rythme et de réaction aux évènements justifie cette imbrication	Barrière de sortie synchronisée sans objet du fait de l'absence de parallélisme. Tout le bloc est considéré comme une instruction par le bloc père qui lui applique en fin sa barrière de synchronisation. Ainsi les membres non actifs attendent l'agent qui exécute le bloc. La différence de rythme et de réaction aux évènements justifie cette imbrication	parallélisme Barrière de sortie synchronisée sans objet du fait de l'absence de parallélisme. La différence de réactions aux évènements justifie cette imbrication

3.4.3 Type label

Le type Label offre un service amélioré au programmeur concernant les invocations asynchrones à l'API de l'agent (*Cf.* 3.4.1). Il donne la possibilité de connaître la terminaison d'un certain nombre d'invocations asynchrones. Son utilisation permet de pallier à deux situations non satisfaisantes :

- Lors de l'invocation asynchrone d'une primitive qui renvoie un résultat, toute utilisation de ce dernier provoque le blocage du programme jusqu'à ce que le résultat de l'invocation soit disponible.
- Lors de l'invocation asynchrone d'une primitive qui ne renvoie pas de résultat, il n'y a pas de possibilité de savoir quand la primitive termine.

Un label permet de savoir que le résultat n'est pas encore disponible afin de différer son utilisation au bénéfice d'autres tâches.

Il faut d'abord le déclarer avant d'utiliser ses services.

Déclaration des labels :

[E1] label_declaration::= **LLabel** labelname_list | **SLabel** labelname_list

Dans un E-bloc, plusieurs labels peuvent être déclarés. Suite à cette déclaration, chaque agent membre du E-bloc peut y accéder par son nom. Un label peut être partagés par plusieurs agents ou local à un agent.
L'instanciation et l'initialisation du label sont implicites et sont à la charge du runtime MASL.

Label local
label_declaration::= **LLabel** labelname_list
Un label local n'est accessible que par un seul agent car il se charge de la gestion du parallélisme interne à l'agent. Il ne concerne que les invocations asynchrones faites par le programme de cet agent. A noter qu'il peut faire un appel asynchrone à une méthode d'un autre agent. Pour un label local déclaré, le runtime a la charge de gérer autant d'instances de label qu'il y a d'agents membres actuels.

Label partagé
label_declaration::= **SLabel** labelname_list
Un label partagé est accessible par tous les agents membres du E-bloc où il a été déclaré. Il concerne potentiellement les invocations asynchrones de tous les agents membres. Ainsi, un agent peut être notifié de la terminaison des invocations asynchrones faites Pour un label partagé déclaré, le runtime a la charge de gérer une seule instance et les accès conflictuels.

Portée d'un label :
Un label est déclaré dans un E-bloc. Un agent membre de ce bloc ou d'un de ces fils à la possibilité de l'utiliser.
Pour un label local, avant que l'agent ne devienne actif, une instance est initialisée : elle ne contient aucune invocation asynchrone.
Pour un label partagé, son unique exemplaire est initialisé avant l'activation du premier membre.
Lorsque l'agent quitte le E-bloc, il perd cette possibilité. Pour un label partagé, si une méthode de compensation (*Cf. 3.1*) est déclenchée automatiquement, l'appel asynchrone correspondant est considéré par le label comme terminé. Pour un label local, l'instance correspondante est détruite à la sortie du bloc.
Pour un label partagé, l'unique exemplaire est détruit à la sortie du dernier membre du bloc.

Problèmes liés :
Dans le cas d'un label partagé, la sortie d'un membre ayant fait des invocations asynchrones n'entraînant pas le déclenchement d'une primitive de compensation peuvent, selon le cas être considérés comme appartenant encore légitimement au label ou non. Ce choix est laissé libre à l'implémentation du runtime MASL.

Services offerts par un label :

<label_name>**.**<agent name>**.**method([parameters,*])
| <label_name>**.this.**method([parameters,*])
| <label_name>**.**<identifier **=** <agent name>**.**method([parameters,*])
| <label_name>**.**<identifier **= .this.**method([parameters,*])
| <label_name>**.**<agent name>.<identifier **=** <agent name>**.**method([parameters,*])
| <label_name>**.**<agent name>.<identifier **= .this.**method([parameters,*])

| <label_name>**.this.**<identifier **=** <agent name>**.**method([parameters,*])
| <label_name>**.this.**<identifier **= .this.**method([parameters,*])

Le premier service offert permet à l'agent d'ajouter une invocation asynchrone. Cette dernière est immédiatement lancée. L'exécution non bloquante peut concerner l'API de l'agent lui-même s'il correspond la référence du destinataire, ou l'API d'un autre agent. A noter que l'instruction consiste à faire un appel non bloquant. Dès que ce dernier est fait, l'instruction termine, indépendamment de la terminaison de la primitive. L'agent peut alors être soumis à une barrière de synchronisation.

| <label_name>**.isFinished()**: boolean

Le deuxième service offert est de connaître la terminaison globale de l'ensemble des invocations du label. Si jamais une invocation n'est pas terminée, **.isFinished()** renvoie `false`. Si toutes les invocations sont terminées, **.isFinished()** renvoie `true`.

<label_name>**.free()**

Enfin, **.free()** permet de réinitialiser le label pour pouvoir le réutiliser. La terminaison des invocations encore en cours sera ignorée dans le programme MASL.

Exemple d'utilisation d'un label local :

Soit le fichier XML qui définit un type d'agent qui a une variable d'instance `currentStepResult`, des primitives non bloquantes `fgetTimeConsumingCalculationResult()` et `fdoTimeConsumingWork()`.

```
01| import SomeType.xml as Sometype;
02| Sometype a0,a1,a2,a3 = newAgent(Sometype);
03| asynchronous entry main (true) {
04|   LLabel llabel;
05|   llabel.this.currentStepResult=.fgetTimeConsumingCalculationResult();
06|   llabel.this.fdoTimeConsumingWork();
07|   while (!llabel.isFinished()){
08|     .log("Doing other task : waiting 1 second…\n");
09|     .sleep(1000);
10|   }
11|   .log("All is done and the result is :"+.currentStepResult);
12| }
```

L'exemple illustre la possibilité de faire des traitements tant que la terminaison des méthodes asynchrones n'a pas été notifiée.

Une version sans label peut être proposée :

```
01| import SomeType.xml as Sometype;
02| Sometype a0,a1,a2,a3 = newAgent(Sometype);
03| asynchronous entry main (true) {
04|   .currentStepResult=.fgetTimeConsumingCalculationResult();
05|   .fdoTimeConsumingWork();
06|   .log("The calculation is done or the agent will wait");
07|   .log(" the result that is ");
08|   .log(.currentStepResult+"\n");
09|   .log("The fdoTimeConsumingWork may be done\n");
10| }
```

Dans cette version, l'utilisation du retour de la primitive à la ligne 8 est bloquante. Le parallélisme interne des agents est peu mis à profit.

Exemple d'utilisation d'un label partagé :

Soit le fichier XML qui définit un type d'agent qui a une variable d'instance `currentStepResult` et des primitives non bloquantes `fgetTimeConsumingCalculation()`, `fdoTimeConsumingWork()`et un fonction bloquante `isSomeType()` qui renvoie systématiquement `true`.

```
01| import SomeType.xml as Sometype;
02| import Default.xml as Default;
03| Sometype a1,a2 = newAgent(Sometype);
04| Default leader = newAgent(Default);
05| asynchronous entry main (true) {
06|   SLabel slabel;
07|   asynchronous entry SomeTypeWork (.isSomeType()) {
08|     slabel.this.currentStepResult=.fgetTimeConsumingCalculation();
09|     slabel.this.fdoTimeConsumingWork();
10|     .log("Doing other task…\n");
11|   }
12|   asynchronous entry SuperUser (.getProxyID()==leader.getProxyID()) {
13|     while (!slabel.isFinished()){
14|       .log("Doing other task : waiting 1 second…\n");
15|       .sleep(1000);
16|     }
17|     .log("All is done.");
18|   }
19| }
```

L'exemple illustre pour un agent, la possibilité d'être notifié de la terminaison de traitements asynchrones demandés par d'autres agents.

3.5 Variables

Types élémentaires :

Les variables ont un type élémentaire universel au sens ou ils sont présents dans tous les langages. Ainsi les types `byte`, `char`, `short`, `int`, `float` et `double` sont disponibles. Les types dépendent du système cible. Par exemple on peut aussi utiliser le type `boolean` si le système cible supporte java.
Chacun des types élémentaires a une valeur par défaut :

- 0 pour `int`, `byte`, `short`, `float` et `double`;
- '\u0000' pour le type `char` ;
- `false` pour `boolean`.

Types composés :

Il est possible d'utiliser des tableaux des types élémentaires si ces derniers sont présents dans le langage cible. Un objet tableau est un objet constitué de composants qui ont le même type qui sont accessibles par un index. Un tableau a des dimensions. Chaque dimension a une limite supérieure fixe.

Portée d'une variable :
Une variable est déclarée dans un E-bloc. Un agent membre de ce bloc ou d'un de ces fils à la possibilité de l'utiliser.
Pour une variable locale, son exemplaire est initialisé avec la valeur spécifiée dans l'expression ou avec la valeur par défaut de son type.
Lorsqu'un agent quitte le E-bloc, il perd son accès à la variable partagée.
Pour une variable locale, l'exemplaire correspondant à l'agent sortant est détruit à sa sortie du bloc.
Pour une variable partagée, l'unique exemplaire est détruit à la sortie du dernier membre du bloc.

Déclaration d'une variable :

Variable locale :

[E2] local_variable ::=**local** type_indication identifier_list [= expression]

Une variable locale n'est accessible que par un seul agent pour peu qu'il soit membre du E-bloc ou elle a été déclarée. A l'entrée du E-bloc, elle est allouée sur la pile du programme de l'agent. A la sortie du E-bloc, elle est désallouée. Si une expression est présente, l'affectation est faite juste après l'allocation.

Variable partagée :

[E3] shared_variable ::=**shared** type_indication identifier_list
[**limited**(agents_type)] [= expression]

Une variable partagée est accessible par tous les agents membres du E-bloc où elle a été déclarée. Pour une variable partagée déclarée, le runtime a la charge de gérer le seul exemplaire et les accès conflictuels.
Parmi les agents membres de son E-bloc, la variable partagée peut interdire l'accès en écriture pour certains types d'agents avec l'utilisation du mot clé **limited** aux agents dans le type. A noter que l'on peut artificiellement différencier des types d'agents pour affiner les accès en écriture des variables partagées.

Initialisation d'une variable :

[**local** | **shared**] type_indication identifier_list [**limited**(agents_type)] = expression.

Elle correspond de fait à l'allocation de la variable puis à son affectation.
Pour une variable partagée, l'initialisation est assurée par le runtime de telle sorte qu'elle n'ait lieu qu'une seule fois.

Affectation d'une variable :

[F2] variable_assignment ::= identifier **=** expression

Pour une variable partagée, l'affectation est assurée par le runtime de telle sorte qu'elle n'ait lieu qu'une seule fois.

3.6 Les évènements et les réactions

Des évènements peuvent s'échanger entre agents. Les événements peuvent être locaux ou partagés. Ils sont définis dans un E-bloc. Un évènement est déclaré. Il est émis. La diffusion de ses occurrences a pour objet une réaction de chaque agent destinataire.

Déclaration d'un évènement :
Un évènement est déclaré dans un E-bloc. Il peut être déclaré selon 2 manières.

local event identifier_list; [D19]
L'évènement déclaré comme local ne sera perçu que par l'agent émetteur. Il permet de faire communiquer des composants internes à l'agent ou il s'agit d'une exception générée par l'API et promue par le runtime en évènement local.

shared event identifier_list; [D20]
Un évènement partagé va être diffusé à l'ensemble des agents membres du E-bloc où il a été défini. L'émetteur le perçoit aussi.

Emission d'un évènement :
L'instruction **emit (**e : event**)** permet à un agent d'émettre un évènement. Seul un agent membre du E-bloc où l'évènement a été défini peut l'émettre. Chaque destinataire perçoit une occurrence. L'information principale portée par l'évènement est son type. Eventuellement un message le complète si l'instruction **emitWithMessage(**e : event, message : String**)** est employée. Dans ce cas, l'agent destinataire peut obtenir ce complément d'information par l'instruction **getMessageFromEvent(**e : event**)** : String lors de sa réaction.

Emission implicites d'évènements locaux systèmes :
Les exceptions lancées lors de l'exécution d'une méthode de l'API d'un agent sont promues en tant qu'évènement local de l'agent appelant. Une occurrence de l'évènement local `uncaught_exception` avec comme complément d'information le nom de l'exception est perçue par l'agent.
La perméabilité (*Cf. 3.4.1.5*) peut interdire des accès en lecture, en écriture, en exécution. Un évènement `forbidden_access` peut être émis dans le programme de l'agent demandeur de l'accès. C'est un choix d'implémentation du runtime.

Réaction d'un évènement

```
[F12] entry_bloc_definition ::=
      entry_mode entry <entry name> (test) { MASL_declaration; *
        MASL_instruction ;*
        [react  (event)          { react_instruction; *} ]*
        [react  (default)        { react_instruction; *} ]
      }
```

A un E-bloc peut être attaché des blocs **react** pour des types d'évènements différents. Ils précisent ce que doit être la réaction de l'agent suite à la perception d'un évènement.
C'est l'occurrence d'un type d'évènement qui déclenche une réaction : l'agent qui déroule instruction par instruction son programme exécute alors la réaction attachée au E-bloc courant pour le type d'évènement perçu d'un seul tenant.
S'il n'existe pas de réaction prévue pour le type d'évènement dans le E-bloc, c'est la réaction par défaut, définie par **react** (**default**) qui est exécutée.
S'il n'existe pas de réaction par défaut, c'est la réaction prévu par le E-bloc père qui s'exécute et ainsi de suite.
S'il n'existe pas de réaction dans les E-bloc pères, l'évènement est ignoré.

file d'évènement à priorité d'un agent :
Autant les instructions du programmes sont réputées avoir une durée, autant la réaction d'un évènement se veut instantanée. L'agent perçoit en continue ses évènements mais il ne peut y réagir qu'à la fin de l'exécution de l'instruction courante (*Cf.* 3.3) de son programme. Entre temps, les évènements perçus par l'agent ont été rangés

dans une file. Après l'exécution d'une instruction du programme, cette file est vidée selon un ordre basé sur la priorité décroissante de l'évènement pour le E-bloc courant. Cette priorité correspond à l'ordre de définition des réactions. La première réaction définie dans le bloc correspond à l'évènement le plus prioritaire. Si plusieurs occurrences du même type sont perçues par l'agent, c'est l'ordre de perception qui définit celui qui sera traité en premier.
Tout évènement perçu après le début du traitement de la file d'évènement voit son traitement reporté à l'instant suivant : à la fin de l'exécution de l'instruction suivante.
La perception d'un évènement est indépendante du E-bloc courant. Par contre c'est la réaction à cette perception qui dépend du E-bloc. Eventuellement, il sera ignoré. Mais le départ de l'agent du E-bloc n'entraine pas la réinitialisation de la pile d'évènements.

Retour au programme de l'agent défini dans la réaction du E-bloc courant :
Lorsque la réaction est totalement exécutée, les réactions aux évènements moins prioritaires sont elles aussi exécutées. Ce n'est que quand la pile d'évènements de l'agent est vide que le retour au programme s'opère.
Lors de la réaction du dernier évènement dans la pile, les instructions suivantes déterminent le retour :
L'instruction **resume** fait exécuter la prochaine instruction du programme. Elle ne peut être présente que dans un bloc react ;
L'instruction **restart** fait exécuter la première instruction du bloc ;
L'instruction **reelect(**1**)** oblige l'agent à réévaluer le test du E-bloc courant. Si l'agent se redétermine favorablement, il exécute à nouveau la première instruction du E-bloc. S'il ne se détermine pas favorablement, il quitte le E-bloc ;
L'instruction **reelect(**2**)** oblige l'agent à réévaluer le test du E-bloc père. Si l'agent se redétermine favorablement, il exécute à nouveau la première instruction de ce E-bloc. S'il ne se détermine pas favorablement, il quitte le E-bloc ;
L'instruction **reelect(**entryname**)** oblige l'agent à réévaluer le test du E-bloc désigné par son nom. Toute fois, le nom doit désigner un E-bloc dont l'agent est membre. Si l'agent se redétermine favorablement, il exécute à nouveau la première instruction de ce E-bloc. S'il ne se détermine pas favorablement, il quitte le E-bloc.
Dans la mesure où il peut être choquant que ce soit l'évènement le moins prioritaire qui détermine le retour au programme de l'agent, l'instruction **clearEvents** permet de vider lors du traitement d'une occurrence d'évènement la pile de l'agent afin de garantir que c'est bien la réaction courante qui détermine le retour de l'agent dans son programme.
L'absence d'une de ces instructions de retour implique le départ de l'agent du bloc courant.
Il est important de noter que les instructions **restart** et **reelect** sont possibles dans un programme de l'agent. Ce ne sont pas une exclusivité de la réaction comme c'est le cas de **resume** et **clearEvents**.

Retour au programme de l'agent défini dans la réaction du E-bloc père :
Si le traitement d'un évènement peu prioritaire est défini dans un E-bloc père, les retours possibles au programme sont :
L'instruction **resume** fait exécuter la prochaine instruction du programme du E-bloc courant;
L'instruction **restart** fait exécuter la première instruction du E-bloc ;
L'instruction **reelect**(1) oblige l'agent à réévaluer le test du E-bloc courant. Si l'agent se redétermine favorablement, il exécute à nouveau la première instruction. S'il ne se détermine pas favorablement, il quitte le E-bloc.

```
01| import Default.xml as Default;
02| Default a1 = newAgent(Default);
03| asynchronous entry main (true) {
04|   shared event sevent; …
05|   emit sevent; …
06|   react (sevent) {
07|   .log("Reaction to an event");
08|   }
09| }
```

Problèmes liés :
Les variables locales du E-bloc sont accessibles dans la réaction.

3.7 Autres instructions

On ne rentre pas dans le détail de ces instructions classiques.

[F11] basic_instruction :==

MASL_if	[G1]
| MASL_loop	[G2]
| MASL_for	[G3]
| MASL_while	[G4]
| **exit (**level : integer**)**	[G5]

[G1] MASL_if ::= **if** (test) **{** MASL_instruction; ***} [else {** MASL_instruction; ***}]**

Une instruction **if** détermine l'exécution d'une ou aucune des suites d'instructions qu'elle contient, suivant la valeur (logique) d'une ou plusieurs conditions correspondantes.

Une expression qui spécifie une condition doit être de type booléen.

Pour l'exécution des instructions correspondant à **if**, il faut que la condition évaluée vale TRUE ;

Pour l'exécution des instructions correspondant à **else**, il faut que la condition évaluée vale FALSE ;

[G2] MASL_loop ::= **do {** MASL_instruction; ***} loop**

Une instruction **do … loop** comprend une suite d'instruction qui est à exécuter de façon répétitive de manière infinie. C'est l'instruction **exit** qui permet de quitter la boucle. Cependant les instructions **restart, reelect** ont aussi cette conséquence.

[G3] MASL_for ::= **for** (variable_assignement; test; expression) **{** MASL_instruction; ***}**

Comme en java, il suffit de préciser le nom de la variable qui sert de compteur (et éventuellement sa valeur de départ, la condition sur la variable pour laquelle la boucle s'arrête (basiquement une condition qui teste si la valeur du compteur dépasse une limite) et enfin une instruction qui incrémente (ou décrémente) le compteur.

Les instructions **exit**, **restart** et **reelect** font aussi quitter la boucle.

[G4] MASL_while ::= **while** (test) **{** MASL_instruction; ***}**

Cette instruction exécute la liste d'instructions tant que la condition est réalisée.

Les instructions **exit**, **restart** et **reelect** font aussi quitter la boucle.

[G5] **exit (**level : integer**)**

L'instruction **exit** permet de quitter une boucle. Le paramètre permet de préciser la boucle à quitter lorsqu'il y a des boucles imbriquées.

Ceci termine la présentation de la syntaxe concrète et de la sémantique informelle du langage MASL. Cette présentation systématique ne doit pas faire oublier que toutes les notions de MASL se rapportent qu'à une seule construction originale, le bloc entry.

Conclusion

Le langage MASL proposé permet la description de systèmes multi agents, des comportements multi-robots à au moins trois différents niveaux : le niveau société, le niveau groupe, le niveau agent :

- le niveau société représenté par le bloc `entry main` permet d'affecter une mission à l'ensemble des agents. Ainsi, il définit un premier niveau macroscopique de mission.
- le niveau groupe représenté par un emboîtement de `entry`, permet de diviser l'ensemble de la société en différents groupes. Il offre au programmeur une vision d'ensemble sur l'activité de la société et de chacun des groupes.
- le niveau de l'agent est obtenu par une exécution locale dans les agents contrôlés via leur A.P.I. déclarées dans un fichier XML.

De fait, on a autant de niveaux que l'on veut en fonction de l'imbrication des blocs entry. C'est une approche fractale où l'on retrouve le tout dans une partie.

Du point de vue des Systèmes Multi Agents, MASL propose les avancées suivantes :

- l'appartenance dynamique à des groupes : les agents qui s'exécutent dans le même bloc `entry` appartiennent à un groupe. Ils peuvent être dans le même groupe père ou s'exécuter dans des bloc `entry` fils différents. Il existe une perméabilité pour les agents qui sont dans exactement le même bloc courant (ils se considèrent comme des collègues), une perméabilité pour les autres agents, une perméabilité pour la communication reflexive. MASL permet de ce point de vue une généralisation de la notion de coopération au sein d'un groupe puisque il ne reduit pas la communication de deux agents, comme AGR par exemple, qui ne peuvent communiquer directement que s'ils appartiennent au même groupe, c'est-à-dire que s'ils appartiennent à des groupes différents, ils doivent communiquer via un agent commun à chacun des groupes. De plus le mécanisme de perméabilité est exogène à l'agent MASL et contribue à une plus grande modularité ;

- la notion de rôle n'est pas explicitée dans MASL. Le programme contenu dans un bloc `entry` effectivement exécuté par l'agent est en fait son rôle. Il peut différer par rapport à un autre membre du groupe en fonction par exemple de l'évaluation locale d'un test (instructions `if`, `while`) ou de la perception d'évènements locaux différents. Du fait de l'hétérogénéité des agents MASL, un bloc entry définit des rôles. MASL embarque les fonctions de rôles dans le programme des agents ;

- l'A.P.I. des agents prend en compte le parallélisme interne de ces derniers. En effet, les appels asynchrones de méthodes permettent d'encapsuler le lancement d'un processus concurrent interne à l'agent. Les appels automatiques aux méthodes de compensation permettent de terminer des processus critiques internes à l'agent. Ainsi les agents MASL peuvent jouer plusieurs rôles en parallèles par chevauchement des rôles. Notons aussi que lorsqu'il exécute les instructions de son programme, il joue son rôle principal courant. Lorsqu'il répond à une demande d'un autre agent, il joue un rôle secondaire. L'A.P.I. de l'agent n'admet qu'un seul objet actif mais permet d'utiliser plusieurs objets passifs ;

- MASL offre aux agents situés, une définition suffisamment précise du rythme des actions à entreprendre pour permettre une coopération efficace. Par exemple, dans le cadre du projet MAAM, le maintien de la structure de la molécule en équilibre nécessite la préservation d'un certain nombre d'invariants. Les `synchonous entry` sont des outils puissants pour ce type d'implémentation.

Du point de vue de la sureté et de la gestion des erreurs, MASL propose les avancées suivantes :

- les appels automatiques à des méthodes de compensations définies au niveau de l'A.P.I. de l'agent en cas de sortie précipitées d'un bloc `entry` offrent une garantie qu'en cas de terminaison brutale, d'une partie de l'application, des procedures spécifiques seront automatiquement lancées pour arréter du calcul qui est lancé en mode future. C'est au niveau de l'API que l'invocation asynchrone pour une

méthode est rendue possible. A charge pour cette dernière d'offrir les moyens de gérer les arrêts précipités de manière cohérente.

- Le traitement d'évènement local à un agent permet de gérer les incidents. Les politiques de reprise possibles sont la reprise du rôle (instruction `resume`), son recommencement (instruction `restart`) ou réaffectation à un groupe (instruction `reelect`) ou l'interruption simple d'un comportement (politique par défaut : sortie du bloc courant), et la propogation de l'évènement au bloc entry père.

- la perméabilité courante filtre les accès extérieurs et garantit que seuls les agents compétents du point de vue de leur rôle peuvent invoquer un service d'un autre agent. Les profils de compétences ont été définis à priori dans l'A.P.I. et l'agent MASL module son accès en fonction de ces derniers ;

- les états de perméabilités publiés dans l'A.P.I. de l'agent définissent les différents modes de fonctionnements autorisés. Leur définition permet de prendre en compte les modes dégradés définis par les constructeurs des robots (de façon statique uniquement).

Le langage MASL unifie plusieurs paradigmes de programmation :
- pour mieux associer robotique collective et SMA ;
- le data parallélisme au niveau société et au niveau groupe;
- le parallélisme de contrôle au niveau société et au niveau groupe;
- la programmation réactive au niveau agent ;
- la communication par mémoire partagée au niveau société et au niveau groupe;
- la communication par envoi de messages aux niveaux sociétés, groupes et agents ;
- la communication par broadcast d'évènements au niveau société et au niveau groupe.

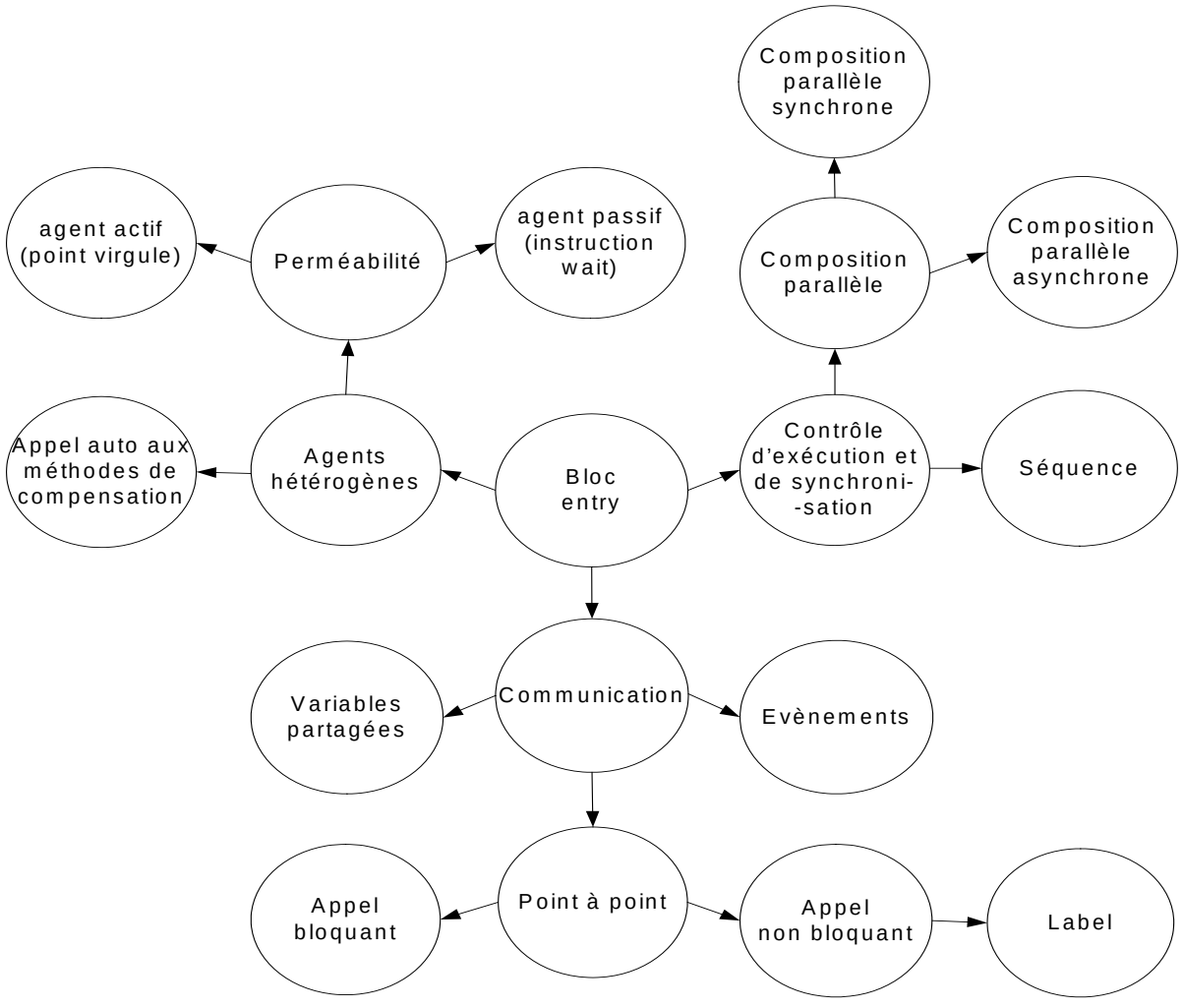


Figure 88 Carte de fonctionnalités de MASL : le bloc entry est au centre.

MASL vise l'extensibilité : le même code peut s'exécuter sur des groupes d'agents de tailles différentes. Pour les composants robotiques, l'extensibilité est sujette à une topologie identique : les programmes MASL peuvent s'exécuter sur des groupes de même topologie au nombre près. Par exemple, la marche d'une rangée MAAM est programmée de la même manière si elle concerne 3 atomes ou 6 atomes. Mais un seul atome ne peut pas constituer une rangée. Pour la robotique en essaim, où l'on ne recherche pas de topologies particulières, il est souhaitable que les performances se dégradent progressivement et non de manière soudaine du fait d'une agrégation émergente plus importante.

MASL peut faciliter le portage d'algorithmes entre composants robotiques.

Perspectives

Nous développons le traducteur automatique de code MASL vers java. Le runtime MASL est pour l'instant centralisé et la traduction se fait à la main. Une fois la traduction faite de manière automatique, la traduction vers le langage c++ ne pose pas de difficultés. Elle devrait aussi faciliter l'expérimentation sur des robots réels.

Les extensions possibles de ce travail

- Nous voulons définir des principes pour une bonne programmation MASL. Une étude approfondie des méthodologies orientées multi-agents est un bon début. Nous nous sommes contenté de souligner la proximité du point de vue MASL avec ces méthodologies. Il reste donc à faire des études de cas comparées des différentes méthodes (GAIA, Voyelles)

- Nous souhaitons doter MASL d'une représentation graphique pour permettre au programmeur une programmation visuelle.

- Nous souhaitons améliorer la sûreté des programmes MASL. Pour cela, une définition formelle de MASL est nécessaire pour permettre de vérifier les programmes MASL, pour prouver des propriétés de ces programmes et permettre leur correction, leur réutilisabilité, leur portabilité et leur introspection.

- Nous souhaitons rendre la plateforme MASL conforme au standard FIPA pour permettre l'interopérabilités avec d'autres plateformes. MASL est focalisé sur les agents robotiques. Une collaboration efficace avec des agents logiciels peut constituer une solution à certains problèmes.

- Nous nous sommes limités à des agents n'ayant pas d'apprentissage ; Il reste à prendre en compte l'utilisation d'apprentissage, de réseau de neurones.

- Nous souhaitons utiliser des environnements robotiques en voie de standardisation comme Microsoft Robotic Studio. [Microsoft Robotic Studio, 07]

Annexe : Vers un modèle d'exécution centralisé

L'objectif de cette annexe est de valider par une expérimentation notre approche par le langage MASL. Dans un premier temps, nous allons étudier la réalisation matérielle du projet MAAM. Puis nous aborderons les deux principaux simulateurs de ce projet. Le runtime MASL est ensuite confronté à plusieurs scénarios de déploiement. Sa spécification est donnée avec un exemple de traduction de programme MASL vers une classe de contrôle Java. Les expérimentations ont portée principalement sur le projet MAAM en simulation comme le montre la Figure 89.

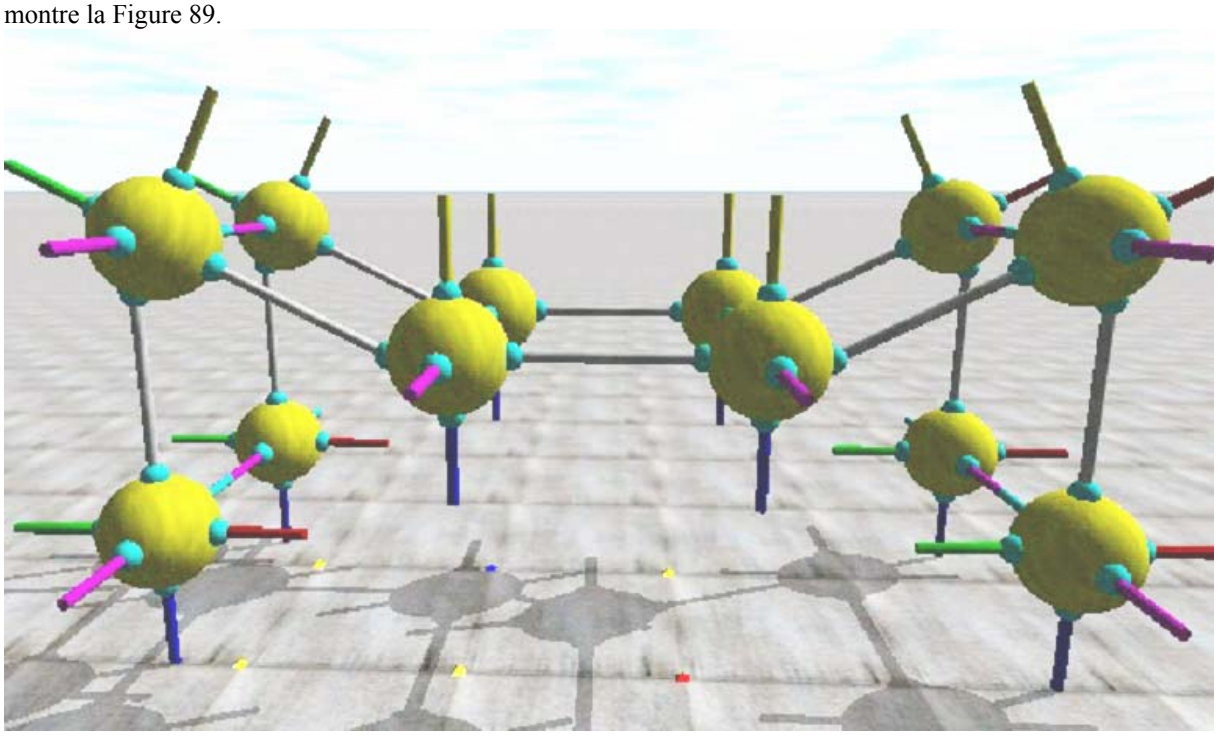


Figure 89 Une araignée MAAM

A.1 Réalisation matérielle incomplète des atomes MAAM

Par rapport au cahier des charges prévisionnel du projet MAAM [Duhaut, 02], une partie a été réalisée et des extensions ont été déjà spécifiées.

A.1.1 Un composant autonome

Claude Guéganno a réalisé la partie matérielle des atomes [Gueganno & all, 04], [Gueganno & all, 05]. Le CPU embarqué dans l'atome est construit autour d'un CSoC (Configurable System on Chip) qui intègre dans le même composant un micro-contrôleur (noyau 8051) et un FPGA (Field Programmable Gate Arrays). Les boucles de bas niveau du robot sont réalisées dans le FPGA.
Compte tenu des caractéristiques de l'atome, le CPU embarqué doit :

1. contrôler 12 degrés de liberté (2 ddl/patte) : chaque patte est actionnée par deux servo-moteurs (moteur courant continu + régulateur PID de position + réducteur). Chaque servo-moteur est commandé par un signal PWM. Les 12 PWM sont réalisés dans le FPGA. Du point de vue du logiciel, commander une patte revient seulement à écrire dans 2 registres. Un registre supplémentaire permet d'activer ou de désactiver chacune des pattes ;
2. identifier les pattes au contact du sol (2, 3 ou 4). Cette information est extraite de l'électronique interne du servo-moteur, en traitant par un monostable redéclenchable les impulsions de commande du hacheur en pont ;
3. commander l'alignement de pattes de deux robots, en vue de l'arrimage: chaque patte est équipée d'un émetteur/récepteur IR (infra-rouge). Un convertisseur AN à 8 voies (max117) est commandé en mode

pipeline par le FPGA, par des trains d'impulsions .Vu du logiciel, il suffit de lire un registre pour obtenir le niveau de réception IR ;

4. communiquer avec un autre atome et/ou un système centralisé de contrôle par liaison radio ;
5. [prévu] actionner 6 pinces. L'actionneur prévu est un matériau à mémoire de forme (muscle filaire) ;
6. [prévu] récupérer et traiter les informations d'un capteur de contact à l'extrémité de chaque patte ;
7. [prévu] récupérer et traiter les informations des trois capteurs d'efforts sur chaque patte ;
8. [prévu] récupérer et traiter les informations des trois accéléromètres et des trois gyromètres du noyau ;
9. [prévu] permettre la communication point à point de voisinage via deux pattes d'atomes différents interconnectées.

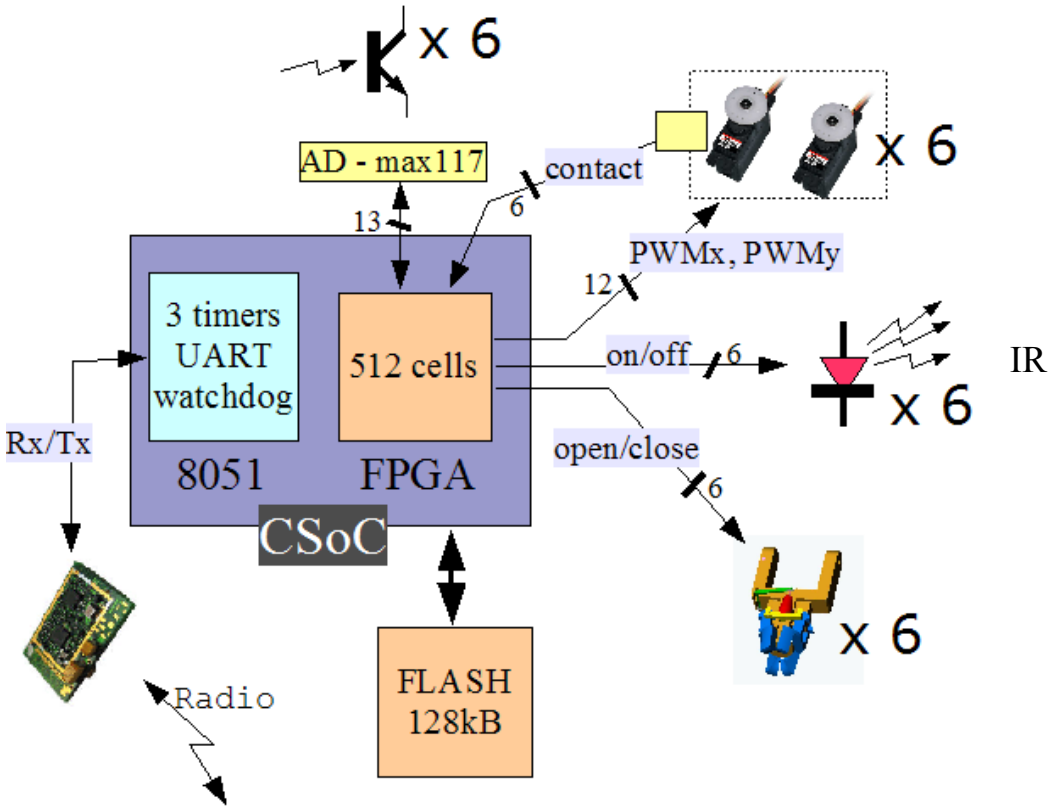


Figure 90 Schéma synoptique de l'architecture matérielle d'un atome

A.1.2 Communication

A.1.2.1 Communication point à point au sein d'une molécule

Que faire quand un atome de la molécule n'est plus joignable ? Un seul atome est joignable et route les messages aux autres atomes de la molécule. On a procédé à une expérimentation du routage de messages au sein des molécules via java RMI. Dans le même esprit, le simulateur centralisé de Claude Guéganno permet à un atome simulé d'interpréter les commandes transmises par un autre atome. L'interprétation dans le cas de la simulation est alors sans changement d'espace de nom. Dans la Figure 92, les flèches représente les communications entre l'ordonnanceur et l'atome élu arbitrairement de chaque molécule. Les traits symbolisent la topologie de la molécule donc les interconnexions entre atomes.

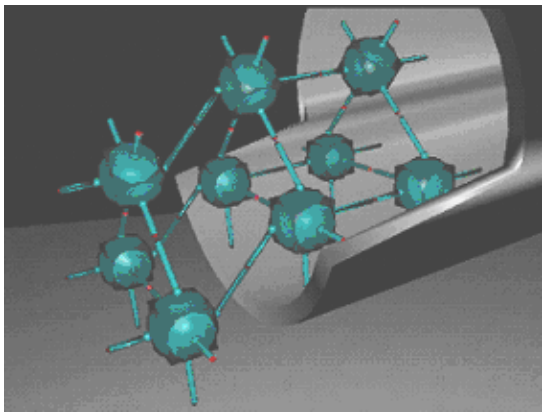


Figure 91 Les atomes hors de portée sont joignables par routage

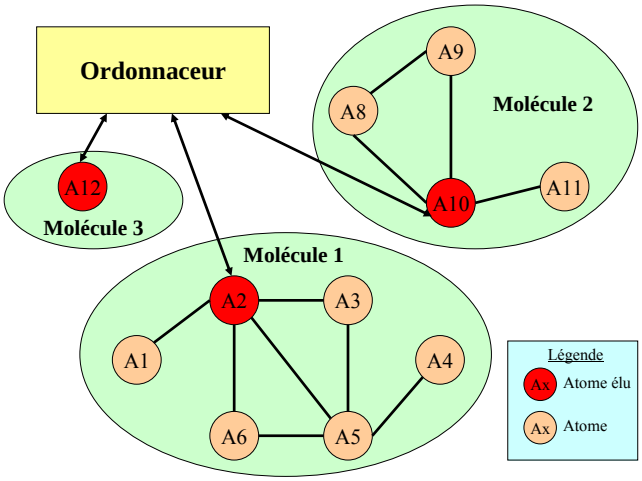


Figure 92 Un atome joignable est arbitrairement élu pour prendre en charge le routage interne à la molécule.

A.1.2.2 Communication bluetooth

Le système de communication ad hoc de MAAM utilise la technologie bluetooth, dont les caractéristiques de coût réduit, de forte intégration des composants et de faible consommation sont déterminantes. Dans un réseau bluetooth, toutes les unités sont identiques, aussi bien pour le matériel que pour le logiciel, mis à part une adresse définie de manière unique sur 6 octets. Le middleware de communication de MAAM est écrit au dessus de la couche HCI (Host Control Interface) de bluetooth, ce qui permet d'exploiter au mieux les possibilités de bluetooth, en particulier la gestion de liens entre un module et plusieurs autres. Les couches bluetooth, de la couche physique jusqu'à la couche HCI sont implantées dans un module industriel construit par Inventel connecté au CPU par une liaison UART.

Il n'y a pas de symétrie dans la communication. Quand un module bluetooth établit une connexion avec un autre, il devient le maître de la communication. Un maître peut avoir jusqu'à sept liaisons ouvertes en même temps. L'ensemble formé du maître et de ses esclaves se nomme un piconet. Un scatternet est le résultat de l'interconnexion de plusieurs piconets. Comme le montre la
Figure 94 Un réseau d'atomes avec bluetooth , les atomes A1, A2 et A3 forment le piconet p1 dont le maître est A1. Les esclaves A2 et A5 sont les maîtres respectifs des piconets p2 et p3. Les huit atomes forment un scatternet.

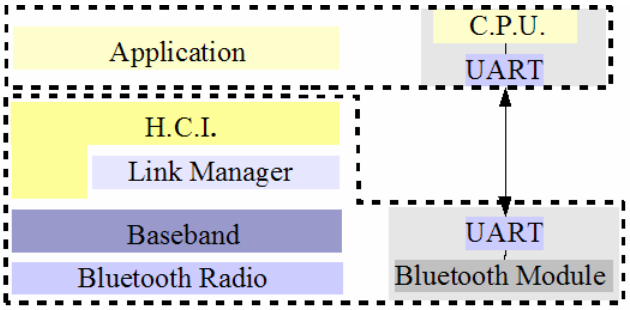


Figure 93 Les couches bluetooth

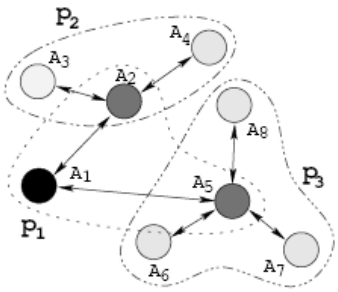


Figure 94 Un réseau d'atomes avec bluetooth

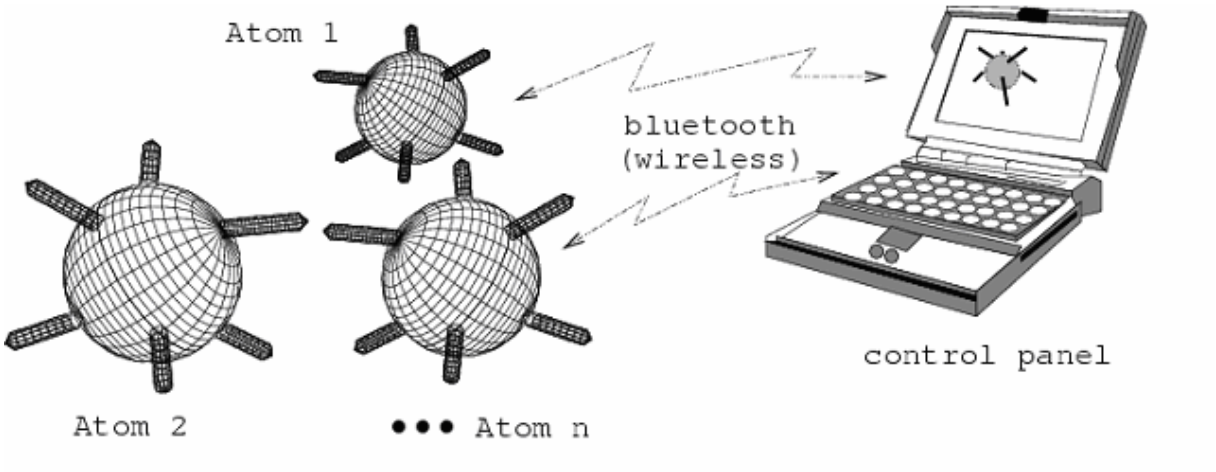


Figure 95 Les atomes sont téléopérés via bluetooth

A.1.3 Un langage interprété de contrôle d'atome

Claude Guéganno [Gueganno & all, 05b] a proposé pour le contrôle local d'un atome un langage interprété. Il comprend des constructions générales comme l'itération (`*`, `while`, `for`), les conditions (`if`, `else`), la parallélisation d'instructions simples internes à l'atome et des primitives déclarées dans une description XML (par exemple `IR_Detect`, `Pas`).
Par exemple si on souhaite que l'atome se déplace jusqu'à ce qu'il détecte un signal infra-rouge, on doit écrire :

```
*[!(Atom.IR_Detecte())](Atom.Pas(););
```

Figure 96 Exemple de commandes internes à un atome.

La Figure 96 montre la traduction en commandes de l'algorithme « TANT QUE tu ne détectes rien FAIRE avancer.

A.1.4 Un mécanisme d'attache partiellement réalisé

Le mécanisme d'attache a été modélisé mais il n'a pas été complètement réalisé. Il s'agit d'un mécanisme d'enclicage passif. Le mécanisme d'attache qui ne nécessite pas de capteurs supplémentaire, est conçu pour recevoir un émetteur et un récepteur infrarouge.
L'attache ambitionne de résoudre de contraintes fortes :

- accrochage passif pour éviter une consommation d'énergie une fois l'accrochage réalisé ;
- n'importe quel patte peut s'accrocher avec n'importe quel autre donc un système hermaphrodite d'attache ;
- l'orientation entre les pattes doit pouvoir être quelconque (tous les systèmes existants on un sens d'accrochage) ;
- ne pas introduire d'actionneur lourd dans le système de verrouillage ;
- avoir un peu de compliance passive pour gérer les incertitudes.

Pour répondre à ces besoins, les principes ont été retenus :

- actionnement par matériaux à mémoire de forme ;
- une seule patte s'attache à l'autre : le système est dissymétrique et le choix de la patte active sera décider par le système de contrôle ;
- un système de pince montée sur ressort sera utilisé ;
- la surface de contact entre pattes sera plane et circulaire ;
- le blocage des pinces sera obtenu par un mécanisme tout ou rien de type stylo à bille.

En position ouverte les mâchoires de la pince sont maintenues verrouillées par le mécanisme de blocage de type stylo a bille. L'actionnement du levier par le matériaux a mémoire de forme libère les mâchoires qui sont poussées à se fermer par des ressorts. Une fois fermés, la pince se présente sous forme d'un cône dans lequel l'autre patte sera guidée lors de la pénétration. Les parties circulaires qui entres en contact au moment de l'assemblage contiendront les émetteurs récepteurs infra rouge permettant la localisation relative des atomes dans l'espace et également une ligne de courant pour permettre un apport externe d'énergie dans la molécule.

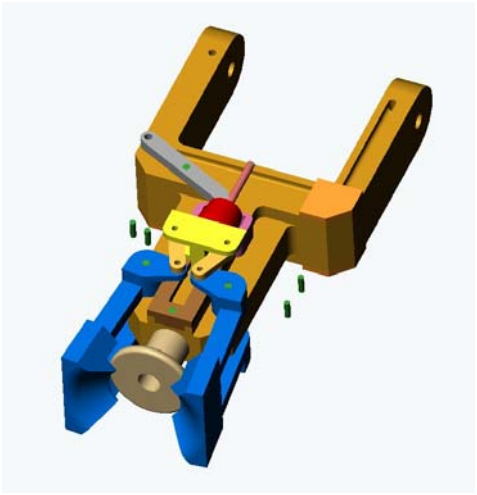


Figure 97 Attache conçue

Figure 98 Deux atomes réels attachés de manière fixe

A.2 La nécessité de simuler

Bien que conçu, le système d'attache n'a put être réalisé complètement. A défaut, la Figure 98 montre deux atomes attachés de manière fixe. Il n'y a pas de possibilité de libérer les atomes, ce qui limite fortement les possibilités de reconfiguration.
Ceci nous amène à privilégier la simulation pour valider notre approche.
La simulation par rapport à l'utilisation de robots réels offre les avantages suivants :

- des coûts réduits ;
- avoir une meilleure sécurité pour les opérateurs ;
- éviter d'abîmer des robots ;
- éviter de consommer de l'énergie ;
- permettre la reproductivité des expérimentations ;
- permettre l'exploitation de traces et fichiers de journalisations ;
- mettre à disposition des environnements complexes ;
- permettre de simuler des pannes ;
- permettre de prototyper des robots ;
- permettre de concevoir des programmes de contrôle qui seront ensuite transférés sur des robots réels ;
- avoir plus de robots que le nombre réel. A fortiori lorsque l'on ne dispose d'aucun prototype ayant les fonctionnalités minimales pour l'expérimentation.

Notre approche se fait surtout dans le cadre des deux derniers points.

A.2.1 Simulation physique

Pour une simulation réaliste, il faut modéliser la physique du monde réel. La Figure 99 montre ce qu'une simulation sans physique peut donner comme aberration (ici, absence de gravité). Une simulation physique comprend un modèle de collision, un modèle de gravité, un modèle de propagation d'efforts, un modèle géométrique. De plus des modèles réalistes de forces, de vitesse et de friction sont nécessaires.
Les modèles traditionnels de la cinématique ignorent la dynamique des forces de systèmes qu'ils modélisent. Ils sont plausibles quand les interactions entre les corps sont limitées (robots à deux roues motrices indépendantes).
Une simulation physique proprement dite utilise la dynamique des corps solides articulés qui modélise l'environnement et les robots comme des ensembles de corps rigides connectés par des jointures. Les corps ne se déforment pas et sont modélisables comme des agrégats de particules qui sont contraintes à se déplacer ensemble. Différents types de jointures sont disponibles : charnières, suspensions, pivots, rotules.
La dynamique tient compte des forces appliquées sur les objets afin de modéliser de manière réaliste les accélérations.
Un corps possède des propriétés constantes (masse, moment d'inertie, coefficient de friction) et variables (position, vitesse, rotation, vitesse angulaire).
Les jointures, les collisions et les forces sont représentées par des équations différentielles décrivant le changement des propriétés variables. Dans le cas général, la solution analytique pour une multitude de corps

n'est pas possible du fait de l'explosion combinatoire. Aussi sont utilisées des approximations numériques. Un des challenges de la simulation physique consiste à obtenir des implantations rapides, stables et précises de ces approximations numériques. La Figure 100 montre le principe de l'intégration telle que présentée par Russel Smith [Smith, 99]. L'intégration calcule les nouveaux états de tous les objets rigides à partir de la description mécanique (jointures entre corps, géométries de chaque composant des corps), de l'état courant de chaque corps, des forces en présence et du paramètre d'intégration qui spécifie l'intervalle de temps écoulé.
Les collisions sont représentées par des jointures temporaires (points de contacts) afin que des forces s'exercent sur les corps en collision. Le but de ces points de contact est de revenir à une situation normale : en effet, pour un moteur de simulation physique des corps rigides, deux objets ne peuvent pas s'interpénétrer (situation anormale) et les points de contact sont donc là pour que les objets se repoussent entre eux lorsqu'ils entrent en collision.
C'est à la charge du programmeur de gérer le "World Stepping" ou intégration : l'avancée du temps dans le monde. Il faut, à chaque état donné, expliquer au moteur physique de combien d'unités de temps il doit avancer les objets. Avancer le temps de une unité de 0.1s n'est pas du tout la même chose que d'avancer le temps de dix unités de 0.01s chacune. En effet, la détection de collision n'est effectuée qu'une fois pour l'ensemble d'une incrémentation du temps, quelle soit de 10ms ou de 100ms. Les collisions et leurs conséquences seraient donc dix fois moins précises lorsque la simulation tourne à 10 ms qu'à 100 ms. Il faut plutôt choisir un "timestep" fixe et à répéter l'avancée du temps autant de fois qu'il le faut entre chaque frame.

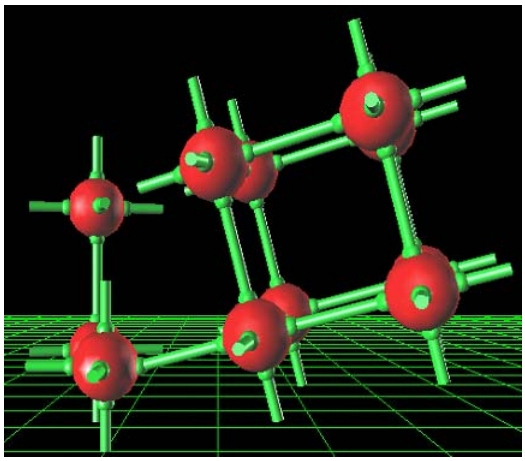


Figure 99 Simulation des atomes MAAM en Java3D sans pesanteur

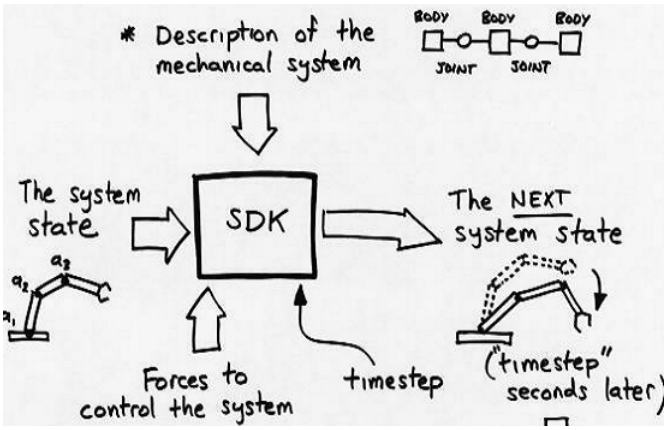


Figure 100 Le principe d'intégration d'un moteur physique d'après Russell Smith [Smith, 99]

A.2.2 Les moteurs physiques

Il existe plusieurs moteurs de simulation physique. Certains sont commerciaux (Vortex [Vortex, 02], Novodex [NovodeX, 02] maintenant connu sous le nom de PhysX [physX, 06], Havok [Havok, 02]), d'autres sont open source comme ODE [ODE, 02] (Open Dynamics Engine). Pour une liste détaillée de moteurs, on peut consulter le site [Trinkle, 02]. D'après l'étude de [Seuling & all, 06], qui prend en compte huit moteurs qui sont gratuits pour une utilisation non commerciale, des différences significatives existent entre ces moteurs. Ainsi Novodex (PhysX), moteur physique parallèle de Microsoft Robotic Studio [Microsoft Robotic Studio, 07], est selon une batterie de test (friction, forces, rebonds, stabilité, extensibilité) le meilleur moteur. Le second qui est très proche en performance est ODE. A noter que Vortex qui ne prévoit pas d'utilisation gratuite ne faisait pas partie de l'étude et est pourtant considéré comme étant le moteur insurpassé au niveau de la stabilité et de la précision.
Pour nous, l'aspect open source prime, seul ODE est donc pris en compte. De plus, il dispose d'une documentation détaillée et constamment enrichie. Il a atteint un niveau de maturité qui permet d'obtenir des approximations numériques précises, rapides et stables. Les méthodes sont basées sur les travaux de Mirtich [Mirtich, 98] pour les collisions avec friction, ceux de Stewart / Trinkle [Stewart & all, 96] et ceux de Anitescu / Potra [Anitescu & all, 97] pour l'intégration et ceux de Barraff [Barraff, 93] pour le solveur LCP (Linear Complementarity Problem). ODE résout les équations des mouvements au moyen de la méthode mise au point par [Cottle & all, 68].
ODE est disponible pour plusieurs plate-formes et utilise une interface de programmation en C pour une plus grande compatibilité, bien qu'en interne, le code source soit écrit en C++. Cette bibliothèque possède plusieurs types de jointures. Le moteur utilise plusieurs intégrateurs en fonction de la précision et de la robustesse de la simulation désirées. Plusieurs primitives de géométries (boites, sphères, cylindres, rayons) sont disponibles et le moteur peut gérer les surfaces constituées de triangles pour discrétiser les géométries continues.

A.2.3 Précision de la simulation

Quel doit être le niveau de précision de la simulation ? Selon le bon sens, le plus fidèle possible à la réalité. Cependant, un robot peut réussir une tâche en simulation et échouer dans la réalité. Pour assurer un transfert efficace de la simulation à la réalité, le corps du robot et l'environnement doivent être reproduits avec soin mais pas forcément avec précision. Cette distinction est expliquée à la suite de ce paragraphe. La Figure 101 montre un s-bot modélisé selon différents niveaux de détail. Chacun utilise des niveaux de ressources de calcul différents.

Dans [Miglino & all, 95], des capteurs bien qu'apparemment identiques d'un robot khepera [Khepera, 02] réagissent très différemment du fait de légères irrégularités électroniques, mécaniques ou de position relative. Il en est de même pour les actionneurs. Pour simuler les irrégularités, les auteurs définissent la notion de bruit de conservation. Il consiste à ajouter une distorsion au niveau des axes x et y de façon à ce que le monde que perçoit le robot ait bougé. Le khepera étant un robot à roue, il n'est pas concerné par l'axe z. Cette distorsion est sensée modéliser les différences d'illumination, la présence d'ombre, de différence minimes entre objets de même type. Dans la Figure 102, les capteurs 5 et 6, bien que disposés du coté droit du khepera de manière très proche réagissent très distinctement.

[Mataric & all, 96] notent que pour mettre au point des programmes de contrôles pour des robots réels en simulation, il est nécessaire de modéliser le bruit et les erreurs afin d'accroître leur transferabilité. Par exemple, en ajoutant un bruit uniformément centré autour de zéro aux valeurs précises produites par les modèles analytiques est la façon la plus simple et commune de faciliter le transfert simulation réalité. Cependant, le bruit dans le monde réel n'est pas uniforme et les robots peuvent encore avoir des difficultés à franchir le fossé entre simulation et ce dernier. L'échantillonnage garantit un meilleur transfert. Les valeurs retournées par les capteurs réels du robot pour des objets donnés et les valeurs des actionneurs à des vitesses données sont mesurées et stockées dans des tables de recherche. Le simulateur accède à ces valeurs et ajoute un bruit. Ainsi, on observe expérimentalement le bruit et plutôt que de le simuler, on utilise les vraies valeurs du bruit.

Jakobi [Jakobi, 98] propose une autre approche. En robotique évolutionniste (combinaison de la robotique, des algorithmes évolutionnistes et des réseaux de neurones), la simulation minimale de Jakobi précise que seules certaines interactions auront lieu si l'évolution réussit… et donc seules celles-là doivent être bien modélisées. Le reste est considéré comme du détail d'implantation et doit être simplifié voire modifié aléatoirement d'un essai à un autre pour que l'évolution ne soit pas polluée par eux. Il faut donc régler la "quantité" de bruit... (bruit blanc i.e. bruit centré sur zéro, bruit de conservation, échantillonnage des capteurs dans le monde réel). Cette méthodologie, qui raccourcit le calcul de l'évolution, suppose que l'on soit capable de définir les caractéristiques à modéliser avec soin.

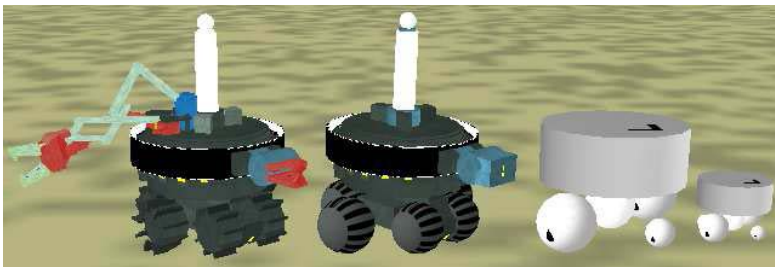


Figure 101 Les différents niveaux de détail d'un s-bot dans le simulateur swarmbot3D [S-bots, 05]

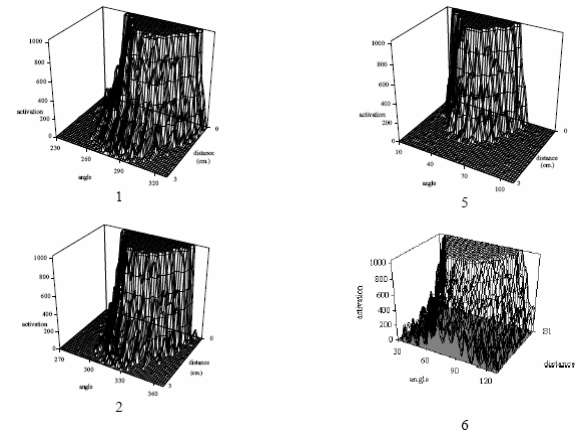


Figure 102 Activation des capteurs IR du khepera

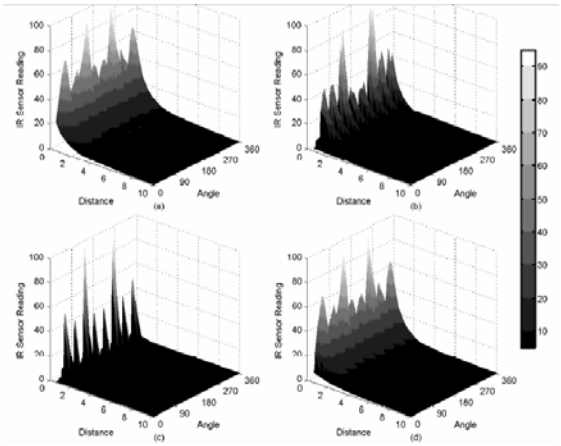


Figure 103 Echantillonage des capteurs de proximité d'un s-bot

La Figure 103 montre les capteurs IR échantillonnés par rapport à un obstacle en fonction d'une distance et d'un angle. Pour (a) l'obstacle est un mur, (b) l'obstacle est un autre robot, (c) l'obstacle est petit et en (d), il est grand. Dans le simulateur swarmbot3d [S-bots, 05], les capteurs sont modélisés à l'aide de ces échantillons et un bruit blanc.

A.2.4 SimulAtom, le simulateur des atomes robotiques (2003-2005)

Le simulateur SimulAtom [Dubois & all, 03], [Simulatom, 04], écrit en C++ a pour but de pallier l'absence d'atomes robotiques réels qui sont développés parallèlement. Le projet MAAM n'a jamais eu pour objectifs de produire en série un grand nombre d'atomes. Aussi dès le début, s'est imposée la nécessité d'un simulateur prenant en compte une population limitée que par la capacité de calculs de l'ordinateur hôte. Il est basé sur ODE mais utilise aussi le framework développé pour le simulateur Übersim [Browning & all, 03], [Ubersim, 02] consacré à la simulation réaliste de robots à roues de la ligue Small-size de la RoboCup.
Ce dernier offre un arbre de scène à ODE et le modèle évènementiel de réaction des agents. Dans ce framework, les agents sont réactifs synchrones [Boussinot & all, 95] (Cf. 1.2.3.3) : un appel à une méthode se traduit par un évènement reçu par l'agent. Avant chaque pas d'intégration ODE, les agents traitent les évènements perçus. Ce traitement consiste à modifier leur paramètres et représentation ODE. La Figure 104 montre que chaque cycle perception/action d'un agent robotique est synchrone avec le calcul d'intégration du moteur physique ODE.

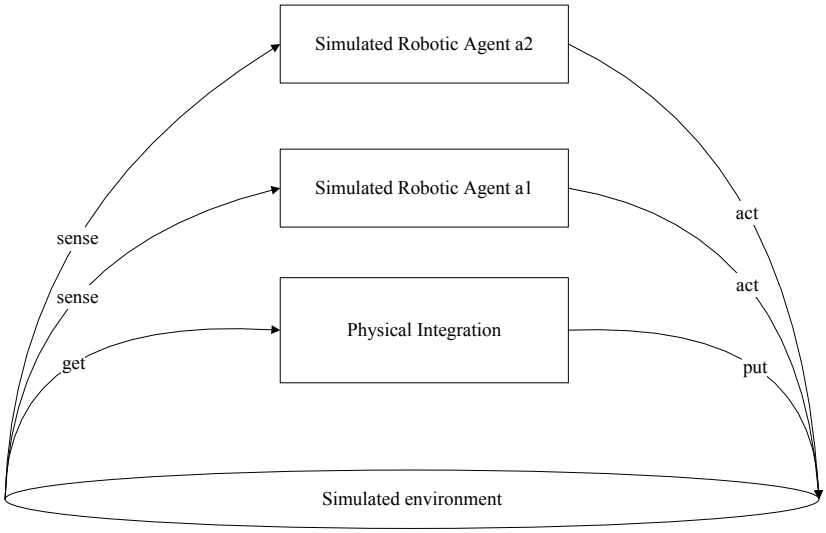


Figure 104 Principe de la simulation dans SimulAtom : les agents simulés sont réactifs synchrones [Boussinot & all, 95]

Nous avons apporté les améliorations suivantes :

- le framework a été étendu aux robots à pattes ;
- le framework a été corrigé pour les configurations cycliques de molécules ;
- ODE a été corrigé pour une meilleur extensibilité : d'abord confronté à une limite de 8 atomes, nous avons réussi à modéliser 125 atomes.
- le rendu des sphères a été amélioré.

Une API de haut niveau [MAAM, 05] a été implantée. C'est l'API représentée à la Figure 107.

Les principales caractéristiques de cette API sont :

- l'appel à l'API est non bloquant. L'appel à un accesseur se contente de la lecture d'une valeur retournée. L'appel à un modificateur permet de fixer une consigne. Puis la main est redonnée au programme appelant. La consigne sera exécutée sur plusieurs pas d'intégration ;
- il faut permettre au programme de contrôle de savoir quand un ensemble de consignes a été totalement exécutées. L'ajout d'une étiquette de synchronisation à la consigne permet de définir les éléments de l'ensemble de synchronisation. Pour connaître si un ensemble de consignes a été totalement exécuté, il suffit de faire appel à la méthode `isFinished()` de la classe `Molecule`.
- il y a une queue à priorité inversée pour mémoriser les différentes consignes. Le mode `KEEP` permet d'insérer la consigne dans la queue. Le mode `REPLACE` vide cette dernière avant l'ajout de la consigne ;
- lors d'un pas d'intégration, si l'objectif de la consigne à exécuter n'est pas atteint, elle est traitée puis elle est ajoutée dans la queue pour être de nouveau traitée prioritairement au prochain pas d'intégration. Sinon, c'est la consigne suivante qui sera traitée ;
- la consigne est étiquetée temporellement pour déterminer quand elle doit être exécutée. Un délai peut être précisé pour retarder son exécution ;

De plus, des classes utilitaires spécifiques à la simulation facilitent la gestion de l'état initial et le débogage de programmes de contrôle.

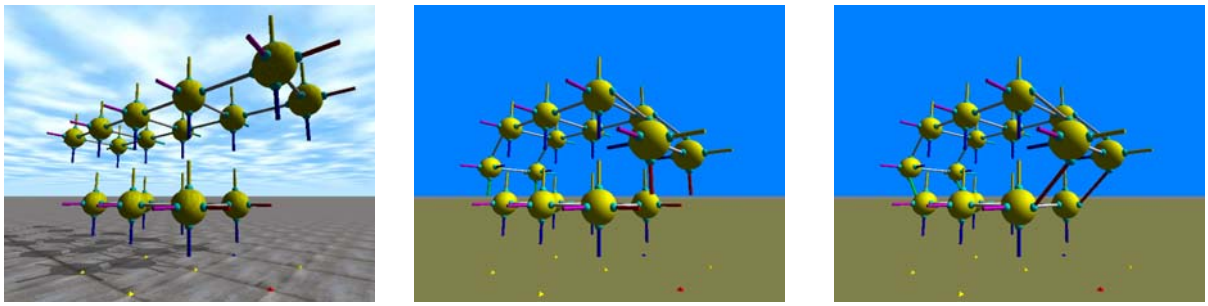


Figure 105 Les étapes de construction d'un état initial cyclique

La Figure 105 montre la construction d'un état initial complexe comme si dans une expérience réelle, un opérateur construisait à priori la molécule. Les atomes sont attachés à l'environnement pour qu'ils ne tombent pas malgré la gravité. Les pas de simulation permettent de prendre en compte les consignes. D'alignés en position canonique (a), les atomes d'en haut modifient l'angle d'une patte (b). Les atomes d'en bas modifient aussi l'angle d'une patte tandis que ceux maintenant au niveau intermédiaire font de même avec une autre patte (c). Les pattes sont alignées, il ne reste qu'à les encliquer pour qu'elles changent de couleur dans la simulation. L'état (d) où les pattes sont encliquées n'est pas représenté.
Le programme de la Figure 106 permet de passer de l'état (a) à l'état (d). Il ne reste plus qu'à détacher les 16 atomes de l'environnement pour que la simulation proprement dite commence. On utilise deux ensembles de synchronisations (`synchro1` et `synchro2`). Dans la mesure où toutes leurs opérations peuvent se faire en parallèle et que la seule contrainte est que les deux soient terminés avant d'encliquer les pattes, la distinction n'est pas nécessaire. Elle a été faite à titre d'exemple.
La méthode `setLink` est en contradiction avec le système d'attache passif défini en 0. Seul la libération des pattes est active. C'est normalement au simulateur de procéder à l'encliquage lorsque les conditions sont réunies. Dans la mesure où on construit un état initial, on est dans le cas où l'opérateur humain force l'arrimage de deux atomes.

```
Unsigned int synchro1=atoms[0]->getMolecule()->getNextSyncIndex();
atoms[3]->getLeg(0)->getServoMotor(0)->setTargetPosition(0,REPLACE,0,synchro1);
atoms[4]->getLeg(1)->getServoMotor(0)->setTargetPosition(250,REPLACE,0,synchro1);
atoms[6]->getLeg(0)->getServoMotor(0)->setTargetPosition(0,REPLACE,0,synchro1);
atoms[7]->getLeg(1)->getServoMotor(0)->setTargetPosition(250,REPLACE,0,synchro1);
atoms[3+8]->getLeg(0)->getServoMotor(0)->setTargetPosition(0,REPLACE,0,synchro1);
atoms[4+8]->getLeg(1)->getServoMotor(0)->setTargetPosition(250,REPLACE,0,synchro1);
atoms[6+8]->getLeg(0)->getServoMotor(0)->setTargetPosition(0,REPLACE,0,synchro1);
atoms[7+8]->getLeg(1)->getServoMotor(0)->setTargetPosition(250,REPLACE,0,synchro1);
unsigned int synchro2=atoms[0]->getMolecule()->getNextSyncIndex();
atoms[0]->getLeg(1)->getServoMotor(0)->setTargetPosition(0,REPLACE,0,synchro2);
atoms[2]->getLeg(0)->getServoMotor(0)->setTargetPosition(250,REPLACE,0,synchro2);
atoms[3]->getLeg(1)->getServoMotor(0)->setTargetPosition(250,REPLACE,0,synchro2);
atoms[7]->getLeg(0)->getServoMotor(0)->setTargetPosition(0,REPLACE,0,synchro2);
atoms[0+8]->getLeg(1)->getServoMotor(0)->setTargetPosition(0,REPLACE,0,synchro2);
atoms[2+8]->getLeg(0)->getServoMotor(0)->setTargetPosition(250,REPLACE,0,synchro2);
atoms[3+8]->getLeg(1)->getServoMotor(0)->setTargetPosition(250,REPLACE,0,synchro2);
atoms[7+8]->getLeg(0)->getServoMotor(0)->setTargetPosition(0,REPLACE,0,synchro2);
…
unsigned int finished=0;
while (finished==0) {
  finished=1;
  for(unsigned int i=0;i<atoms[0]->getMolecule()->getSyncIndexCount();i++)
    if (atoms[0]->getMolecule()->isFinished(atoms[0]->getMolecule()
      ->getSyncIndex(i))==false)
      finished=0;
}
atoms[3]->getLeg(1)->setLink(atoms[0]->getLeg(1));
atoms[7]->getLeg(0)->setLink(atoms[2]->getLeg(0));
atoms[3+8]->getLeg(1)->setLink(atoms[0+8]->getLeg(1));
atoms[7+8]->getLeg(0)->setLink(atoms[2+8]->getLeg(0));
```

Figure 106 Un programme de construction d'une molécule par appels à l'API MAAM

La classe `Molecule` a été conçue dans l'optique de la mission du groupe. Elle offre comme service l'accès facilité à un ensemble d'informations de haut niveau d'intérêt pour l'utilisation de l'Intelligence Artificielle. En effet, SimulAtom est le simulateur officiel du projet Robea MAAM (2002-2005) qui impliquait des chercheurs des équipes des laboratoires GREYC (Université de Caen – CNRS), CEA Paris, LRP (Université Paris 6 – CNRS), LIP 6 (Université Paris 6 – CNRS), LESTER (Université de Bretagne Sud) et VALORIA (Université de Bretagne Sud).

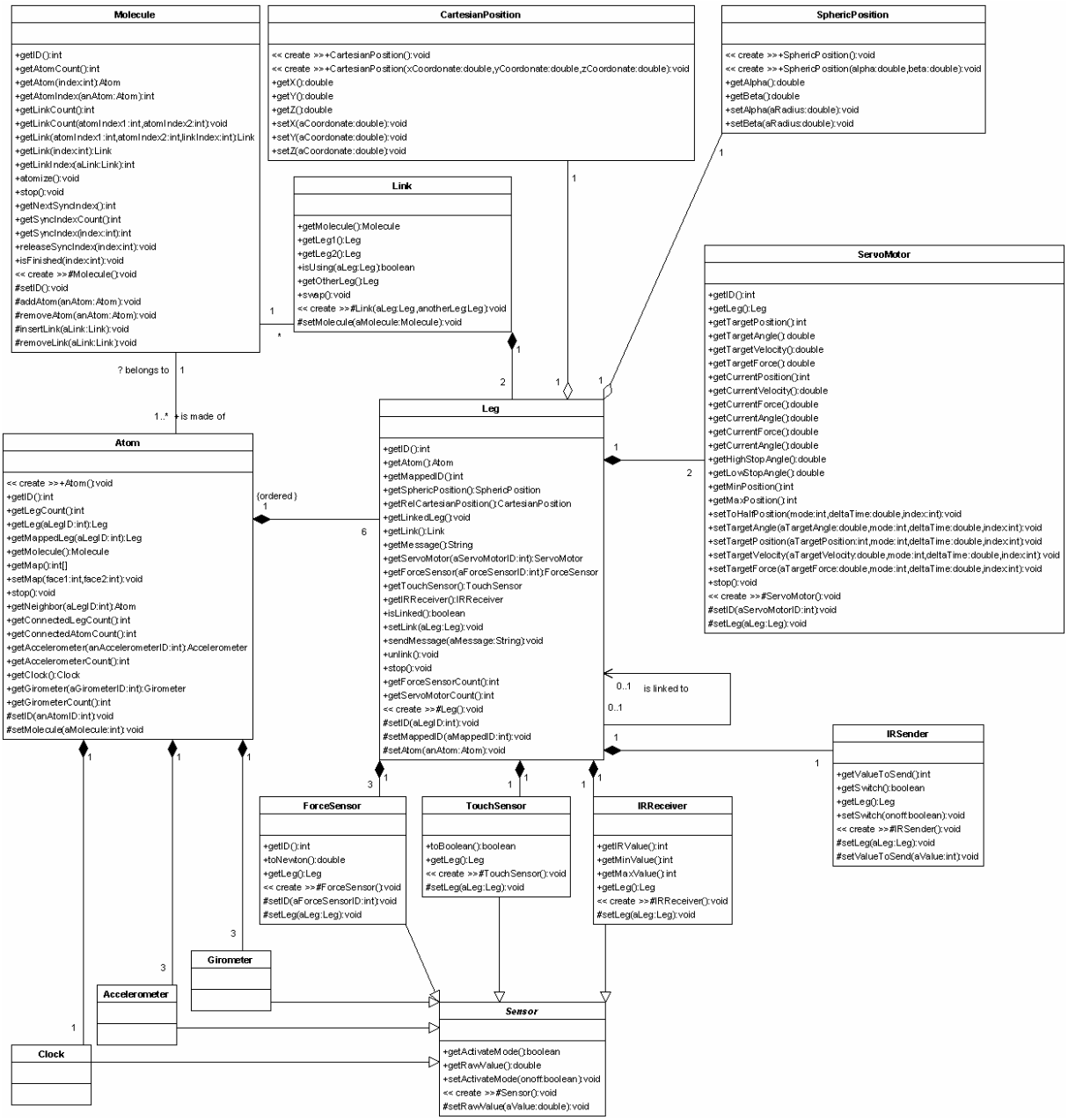


Figure 107 API de contrôle du simulateur SimulAtom

A.2.5 SimExecBots, le simulateur multi-robots (2006-2009)

Le simulateur SimExecBots est construit à l'aide d'une interface JNI [JNI, 97] (Java Native Interface) de ODE [ODE, 02], nommée ODEJava [ODEJava, 04] et du moteur de rendu 3D Java3D [Java3D, 00]. JNI permet à Java d'appeler des méthodes écrites dans un langage natif, C++ notamment. Avec ODEJava, le simulateur offre entre autre la possibilité de charger et d'exporter une description physique d'un agent au format XODE [XODE, 04] (the XML Open Dynamic Engine data interchange format), dérivé de XML. A la différence de son prédécesseur, il se veut multi-robots hétérogènes.
La spécification XODE prévoit des extensions étrangères à la simulation physique. Le simulateur étend XODE au niveau des couleurs du corps et des textures.
Les fichiers au format XODE doivent être traités par un *parser*. Celui livré avec ODEJava a été enrichi pour prendre en compte les géométries rayons, les trois types de cylindres (aligné sur l'axe Z à terminaison plate, aligné sur l'axe Z à terminaison capsule, cylindre utilisateur aligné sur l'axe Y à terminaison plate) et les triangles (Figure 109) ainsi que toutes les jointures (ajout de la jointure universelle, de la rotule et de la rotule motorisée). L'exportation au format XODE a aussi été ajoutée.

A.2.5.1 Les types d'agents

Les agents sont des agents actifs qui possèdent leur propre flot de contrôles. Ils dérivent de la classe Thread. Ils agissent de manière indépendante par rapport à l'intégration du moteur physique ODEJava (Figure 108).
Le simulateur distingue deux types d'agents, les agents logiciels et les agents robotiques. Les agents logiciels sont de simples threads possédant des attributs et un programme de contrôle qui leur permettent d'influer sur la simulation tandis que les agents robotiques partagent ces caractéristiques avec en plus une représentation physique dans la simulation.

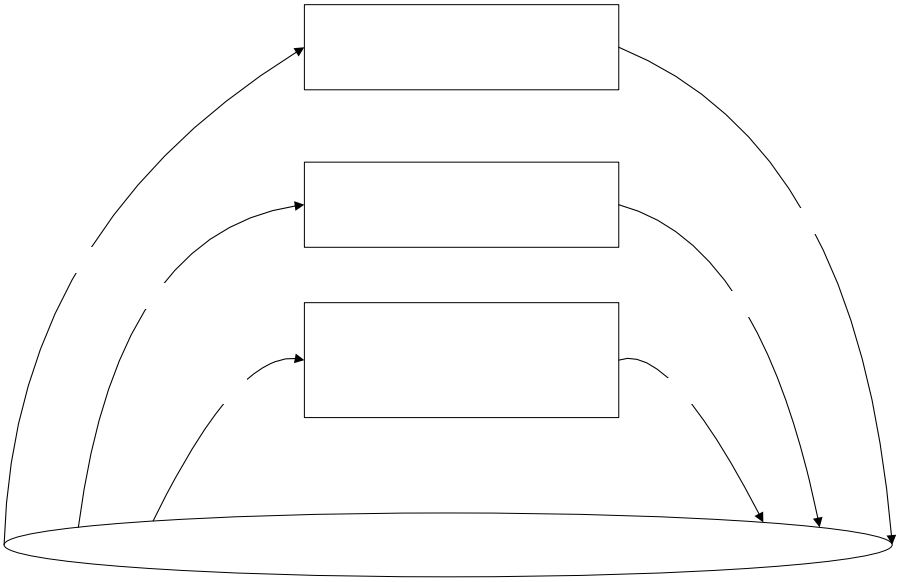


Figure 108 Principe de fonctionnement de SimExecBots : les agents robotiques sont des agents actifs.

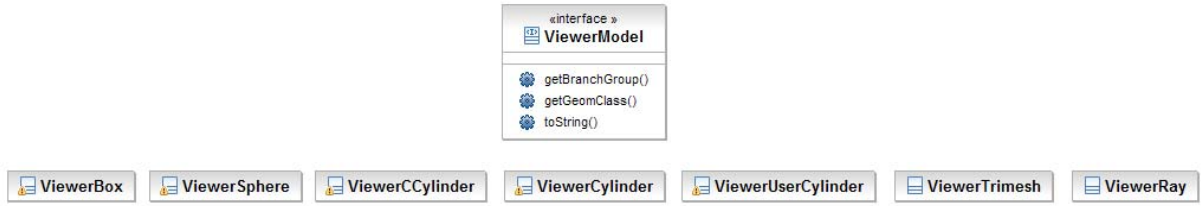


Figure 109 Les géométries ODE représentées dans SimExecBots

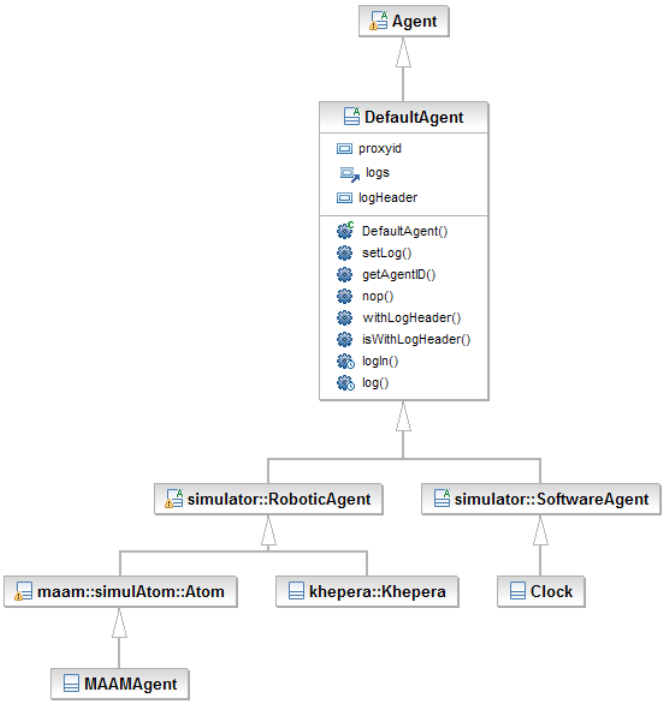


Figure 110 Les types d'agents dans SimExecBots

Les agents logiciels doivent hériter de la classe `SoftwareAgent` et les agents robotiques de la classe `RoboticAgent`, ces deux classes sont abstraites et héritent de la classe `Agent`.

Un agent robotique est d'une part composé d'une représentation physique dans la simulation et d'autre part un programme qui est sensé piloter cette représentation physique.

La partie physique de l'agent est construite à l'aide de l'API d'Odejava. Il est possible de la construire dans une classe java, mais un des avantages du simulateur est de pouvoir associer un fichier XODE à l'agent robotique.
Pour construire le robot, il suffit de réaliser un fichier XODE, puis dans le scénario l'associer à la classe (voir la partie agents robotiques de la construction d'un scénario).
Les `body`, `geom` et `joint` spécifiés dans le fichier XODE sont renommés avec le nom de package, le nom de classe et l'identifiant de l'agent. (Exemple : `agent.Khepera0chassis`) afin que chaque élément de l'arbre de scène Java3D ait un identifiant unique.
Des exemples de fichiers XODE sont disponibles dans la distribution, en plus de ceux fournis avec la spécification XODE.

La classe `Agent` est une classe abstraite qui implémente `Runnable` sans définir la méthode `run()`. Tout agent doit donc définir cette méthode, c'est d'ailleurs dans cette méthode qu'est contenu le programme de contrôle de l'agent. Dans ce dernier, il est fait appel à l'API de l'agent et au run-time MASL.

A.2.5.2 Initialisation de la simulation

Mettre la simulation dans un bon état initial peut s'avérer dans certains cas une tâche très difficile si on n'utilise que les outils vus précédemment. Par exemple, pour effectuer des rotations sur des agents, il faudrait définir un nouveau fichier XODE pour chaque rotation différente.
Le simulateur propose un outil permettant de simplifier la mise en état initiale de la simulation. Les instructions d'initialisation sont exécutées une et une seule fois après l'instanciation des agents et la mise en place de l'espace de variables partagées et avant le lancement des programmes des agents.
Les instructions d'initialisation sont modélisées par une classe java qui hérite de la classe `Initialization`. Cette classe possède des attributs et des méthodes lui permettant de remplir ses fonctions notamment une liste des agents (sur lesquels elle peut par exemple invoquer des méthodes prévues uniquement pour l'initialisation).
Quand l'initialisation est terminée, un thread par agent est créé et démarré pour exécuter la méthode `run()`.
Dans le cadre du projet MAAM, une application Java3D permet de définir l'état initial d'une molécule.

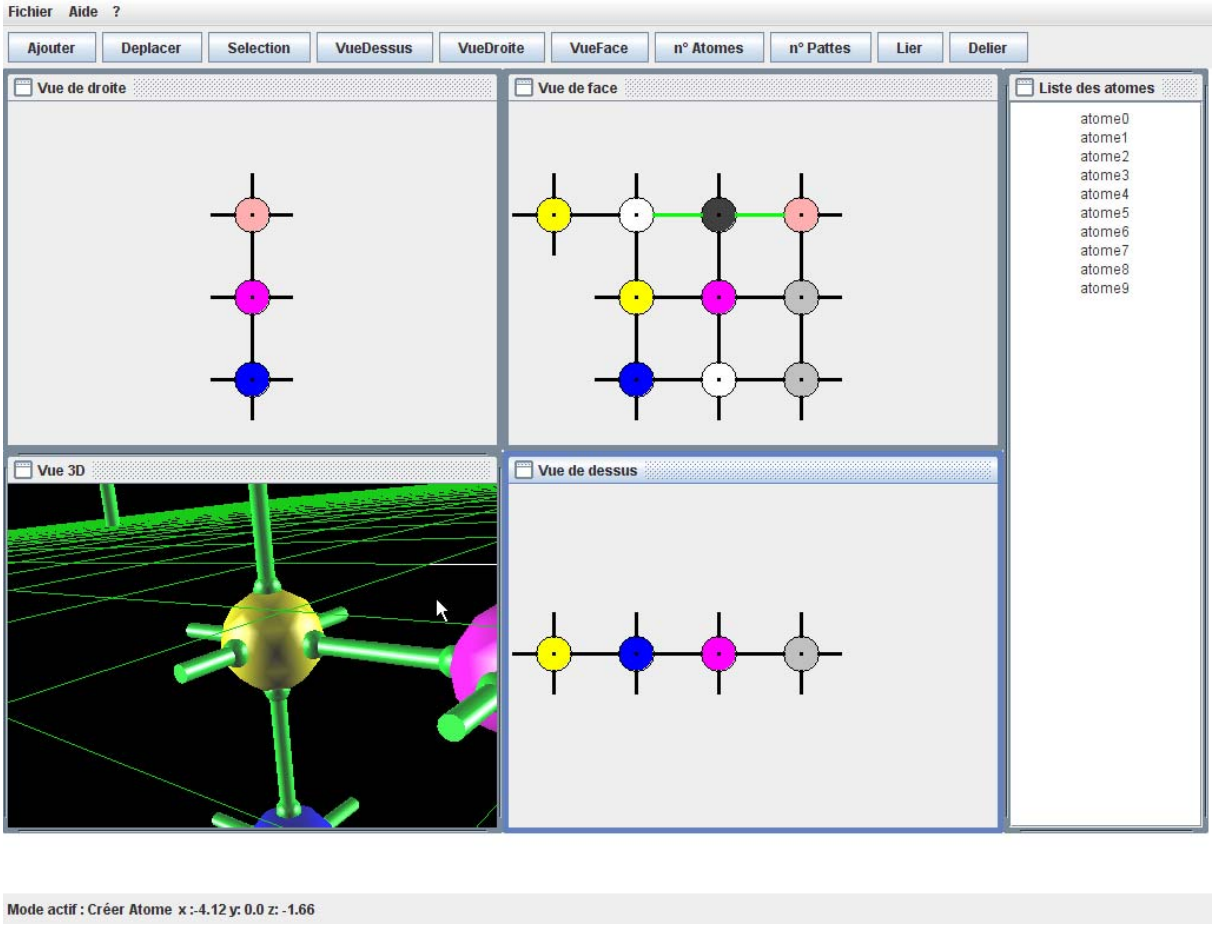


Figure 111 Définition d'un état initial d'une molécule (Java3D)

A.2.5.3 Scénario de simulation

Un scénario donne au simulateur, du point de vue la simulation :
- les fichiers à utiliser pour l'environnement de simulation ;
- le nombre d'agents, les classes à utiliser pour les piloter et si il y a lieu les fichiers les décrivant physiquement et leur définitions de la perméabilité MASL ;
- des instructions d'initialisations qui permettent de mettre la simulation dans un état initial avant de démarrer les programmes des différents agents.

Un scénario donne au runtime MASL du simulateur :
- la liste des blocs `entry` avec notamment leur type (asynchrone, scalaires, synchrone) et leur imbrication ;
- pour chaque bloc `entry`, la déclaration des variables partagées et locale, la déclaration des évènements locaux et partagés, et celle des labels locaux et partagés.

```
<?xml version="1.0" encoding="UTF-8"?>
  <simulation>
    <comment>Exemple de XML de simulation</comment>
    <world file="data/ground.xode" />
    <world file="data/wall.xode" />
    <agent x="0" y="0" z="0" src=" agent/MAAMAgent.java" xodeFile=" data/atom.xode"
     definition="MAAM.xml" />
    <agent src="agent/Clock.java" definition="Clock.xml" />
    <initialization src="simulation/Init.java" />
  </simulation>
  <mission>
    <comment>Exemple de XML de simulation</comment>
    <entry name="main" type="SYNCHRONOUS" immediateRelease="true" logPath=".">
      <sharedvar key="Variable1" value="10" type="int"/>
      <localvar key="Variable2" value="1.2" type="float"/>
```

```
      <sharedevent key="Event1" />
      <localevent key="Event2"  />
      <sharedlabel key="label1" />
      <locallabel key="label2"  />
      <entryname="e1" type="ASYNCHRONOUS" immediateRelease="true" logPath=".">
      </entry>
    </entry>
  </mission>
```

Figure 112 Scénario de simulation dans le simulateur SimExecBots

Dans la Figure 112, on construit un environnement complexe qui est l'union d'un terrain particulier et d'une arène éventuellement close. Deux agents sont instanciés : un agent MAAM à une position déterminée et un agent logiciel. Certains réglages avant simulation sont effectués dans la classe `simulation.Init`. Du point de vue de MASL, deux `runtime.Entry` sont instanciés avec leurs déclarations respectives concernant les variables, les évènements et les labels.

A.2.5.3 Les robots, capteurs, actionneurs simulés

Les robots simulés sont des robots dont leur API de contrôle est publique et qui ont déjà dans un autre simulateur open source basé ODE une implantation. Outre l'API MAAM [MAAM, 05], le khepera possède une API de contrôle java [Khepera, 02] simulée dans [lpzrobots, 06] et dans [Simulator-bob, 03]. Le s-bots du projet swarmbot a été initialement simulé (Swarmbot3D [Pettinaro & all, 03]) à l'aide du moteur physique Vortex. Avec l'extension KODEX [KODEX, 05] (Kovan ODE eXtension) qui permet de charger des robots modélisés sous le système Vortex [Vortex, 02] dans ODE, Swarmbot3D a pu être converti sous ODE et donne accès à l'API publique du s-bot [S-bots, 05]. Toujours avec KODEX, l'API du Kurt3D [Kurt3D, 06] est disponible dans le simulateur MacSim [Ugur, & all, 06]. La simulation d'un robot SONY AIBO ERS-210 est possible avec le simulateur basé ODE SimRobot [Laue & all, 05]. Pour un ensemble complet de capteur, le projet open source Gazebo [Gazebo, 03] est aussi une source d'inspiration.

A.3 Run-time MASL et traduction MASL vers plateforme cible

Nous allons décrire la chaîne de production des programmes de contrôle à partir d'un programme MASL. En entrée, en plus du programme MASL, on trouve le fichier de type de l'agent au format XML (*Cf. 3.4.1*). A partir d'un programme MASL, il existe un algorithme de réécriture pour produire un code source pour un langage de programmation robotique cible. Cet algorithme est en cours de développement. Pour java (Figure 113), le traducteur utilise la définition de type XML et le programme MASL pour générer la boucle réactive/délibérative. Le résultat de cette traduction peut alors être déployé sur une plateforme robotique compatible java dotée d'un runtime MASL ou sur le simulateur Java3D/ODEJava lui aussi doté d'un runtime MASL java. Les deux versions du runtime peuvent différer pour de multiples raisons qui seront abordées au paragraphe suivant.
Si par exemple, la plateforme robotique cible est compatible C++ (Figure 114), il faudra fournir un runtime MASL C++ et un traducteur MASL vers C++. Le codage de runtime ne nécessite pas de gros efforts, une fois le traducteur vers Java et le runtime Java développés dans la mesure où il n'y a pas de difficulté insurmontable. Pour la simulation, on utilise le même simulateur car tous les agents hétérogènes doivent partager le même environnement simulé. Pour faire coopérer des processus C++ et java, il existe des solutions techniques (sockets TCP, pont JNI). Cependant dans la mesure où il faut simuler l'API de la cible dans SimExecBots, la traduction du programme de contrôle MASL en java est une solution préférable.

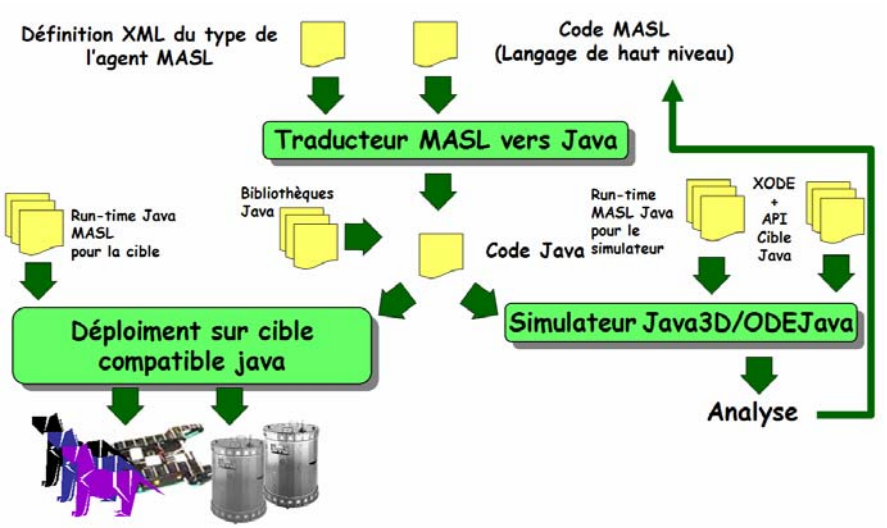


Figure 113 Principe de la traduction d'un programme de contrôle MASL vers une cible compatible Java

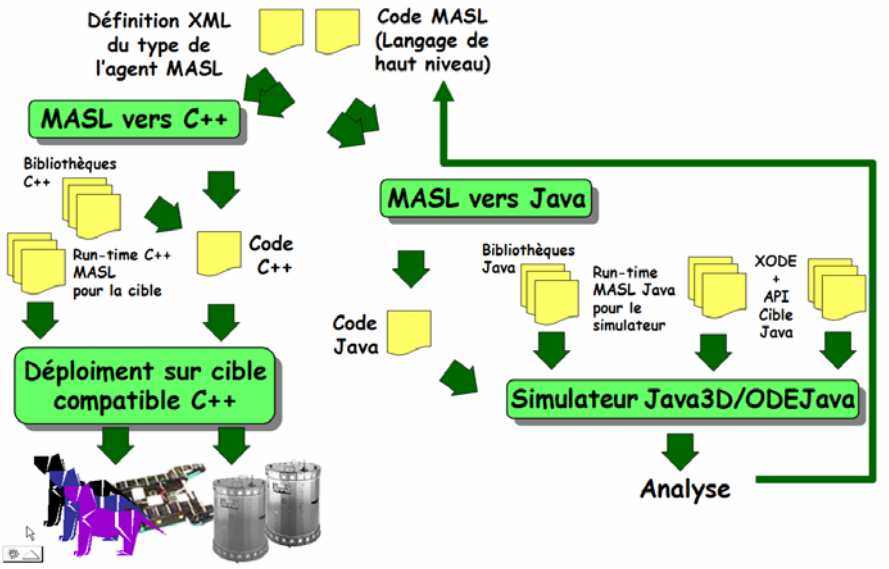


Figure 114 Principe de la traduction de MASL vers une cible compatible C++

A.4 Scénarios de déploiement

Plusieurs scénarios de déploiement ainsi que des modèles d'exécution montrés en Figure 115 ont été envisagés. Pour l'instant, seul les agents simulés (à l'extrémité droite) sont fonctionnels. L'implantation de certains concepts MASL (`entry` synchrones, diffusion d'évènements, variables partagées, blocs `react`, politique `resume`, `reelect`) dépend du modèle d'exécution cible (modèle centralisé ou distribué). A part le cas du robot téléopéré, le programme de contrôle résultat de la traduction du code MASL est déployé sur le robot. Dans le cas du modèle d'exécution distribué, la diffusion d'évènements, l'accès aux variables virtuellement partagées et physiquement répliquées et les mécanismes de synchronisation doivent être implantés au dessus du réseau. Il faut aussi garantir la consistance des accès aux variables partagées.
Nous avons définis des fonctionnalités qui sont dans tous les cas dans chaque agent. Elles seront alors implantées dans le runtime local. Le complément sera géré soit au niveau du runtime centralisé soit au niveau du runtime distribué en fonction du scénario de déploiement.
La boucle délibérative communique uniquement avec le runtime local qui utilise optionnellement des services partagés. Cette règle a l'avantage de rendre le traducteur MASL vers le langage cible indépendant du cas de déploiement.
D'un point de vue des architectures logicielles des agents, seulement quelques fonctionnalités des agents MASL ont une nature délibérative. Par exemple, les variables partagées sont de natures délibératives. En opposition, la diffusion d'évènements partagés est de nature réactive. Les programmes de contrôle avec variables partagées vont donc appartenir à la boucle délibérative. L'agent MASL sera alors hybride. Dans le cas où le programme de contrôle ne comprend aucune fonctionnalité délibérative, l'agent MASL sera réactif. Dans la Figure 115, il faudra alors remplacer les mots « hybrid » et « délibérative » par « reactive ».
Du point de vue de la communication entre les différentes couches applicatives, le robot télé opéré (*remotely connected hybrid agent*) est composé de deux parties : une partie mécanique et électronique aux ordres (passive du point de vue de la décision) qui interagit avec l'environnement et un proxy-agent déporté sur une station de travail ayant une capacité de calcul hors de portée du robot. Une Proxy-API se charge de traduire les appels à l'API publique officielle en messages généralement textuels sur le réseau selon un protocole propriétaire. Par exemple, pour le cas du khepera, l'appel à la méthode de l'API publique `agent.khepera.Khepera.openGripper()` se traduira par l'envoi au robot de la chaîne de caractères `"T,1,D,0"`. Et le robot khepera devra alors répondre pour notifier une bonne exécution par la chaîne de caractère `"t,1,d"`.
Pour la communication entre le runtime centralisé et le runtime local d'agents s'exécutant sur la même machine, il est logique que ces couches applicatives soient écrites dans un langage commun en l'occurrence Java. Notamment en ce qui concerne le simulateur. Eventuellement pour un robot téléopéré, il peut être piloté par un processus C++ par exemple. Un pont JNI, en encapsulant le code binaire C++ dans du bytecode Java, permet de communiquer avec le runtime centralisé Java. Il se charge des conversions notamment au niveau des types des variables partagées et, en ce qui concerne les API, au niveau des attributs, des résultats et paramètres des méthodes. La définition des fichier XML des agents (*Cf. 3.4.1*) tient compte des limites de SWIG [SWIG, 95] (Simplified Wrapper and Interface Generator), un générateur automatique de code JNI (et vers d'autres langages) à partir du langage C/C++.
L'appel à distance de méthodes RPC (Remote Procedure Call) est nécessaire. Plusieurs solutions sont possibles (CORBA, Java RMI, DCOM de Microsoft, XML-RPC, SOAP) pour rendre les accès distants transparents pour l'appelant. Cependant notre préférence est XML-RPC qui a été testé sur la plateforme robotique GdRBot [de Rivera & all, 05]. C'est une spécification et un ensemble d'implantation qui étendent le RPC pour permettre l'appel de procédure au dessus d'Internet afin d'atteindre des machines ayant des environnements d'exécution et des systèmes d'exploitation potentiellement différents. Pour le transport, le protocole HTTP est utilisé et le langage XML permet d'encoder l'appel. De conception simple, XML-RPC est suffisamment puissant pour permettre à des structures de données complexes d'être transmises, traitées et retournées. Il y a beaucoup d'implantations disponibles dans des langages variés (C/C++, Java, Perl, et Python) pour des systèmes d'exploitation variés (GNU/Linux, Microsoft Windows, et Sun Solaris). Ainsi simplicité et portabilité sont les avantages principaux de XML-RPC.

Une plateforme abstraite permet de déterminer le noyau du système en terme de services. Pour passer à une plateforme cible concrète, il y a des ajustements. L'objectif de cette plateforme abstraite est de minimiser ces ajustements lors de l'implantation d'une plateforme concrète.
Pour cela, on définit un modèle d'exécution abstrait qui doit prendre en compte :

- la communication par envoi de messages point à point prenant en compte les appels bloquants, les appels non bloquants utilisant les futures, des labels et des méthodes de compensation ;
- la communication par mémoire partagée gérée à l'aide de sémaphores ;
- la communication par évènements ;

- les exceptions promues en évènements locaux ;
- le synchronisme géré par des barrières d'entrée, de synchronisation et de sortie ;
- les droits gérés par la perméabilité ;
- les groupes hiérarchiques des agents gérés par abonnement/désabonnement.

On peut ranger ces éléments par rapport à leur nature privée ou partagée.

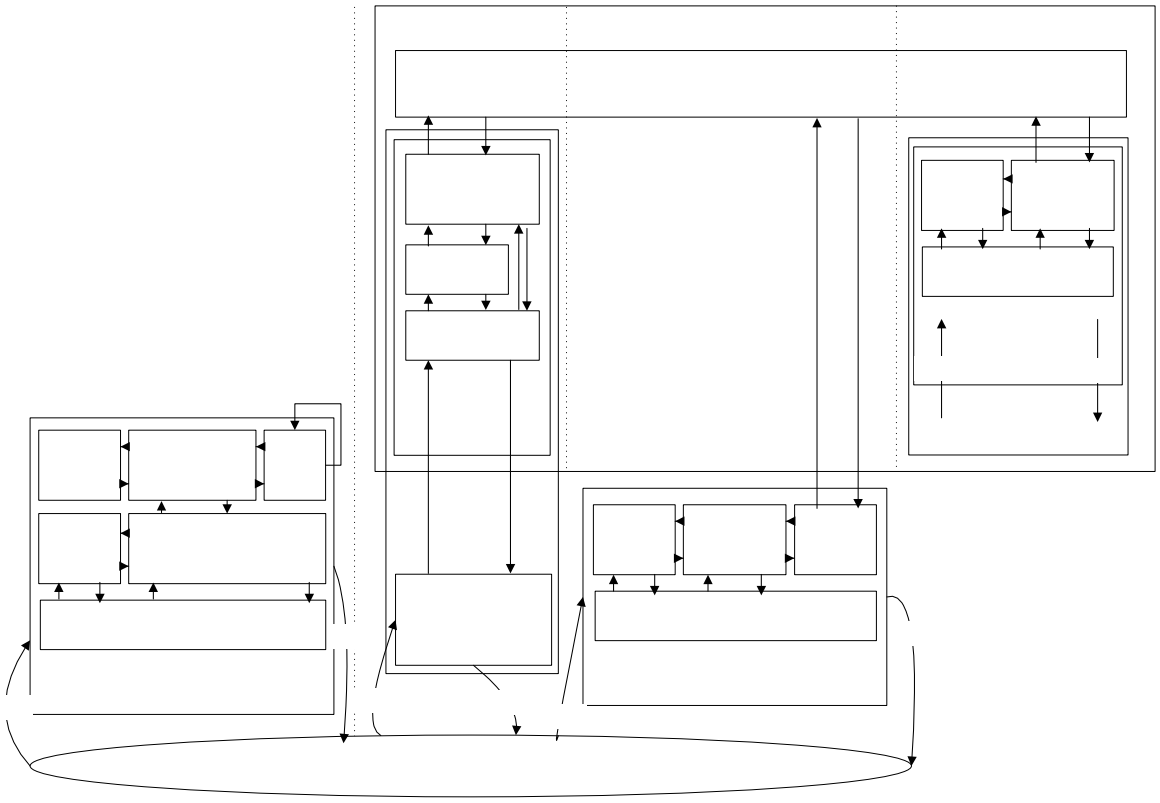


Figure 115 Scénarii de déploiement

Les fonctionnalités privées du runtime sont :

- la gestion des invocations internes des méthodes et des utilisations internes des attributs des agents avec les labels locaux, les futures internes, les méthodes de compensation ;
- la gestion des variables locales ;
- la gestion des évènements locaux et la promotion des exceptions ;
- la réaction aux évènements perçus ;
- la gestion des droits.

Les fonctionnalités partagées du runtime sont :

- la gestion des invocations externes des méthodes et des utilisations externes des attributs des agents avec les labels partagés, les futures externes ;
- la gestion des variables partagées, des variables statiques de classe de l'API ;
- l'interrogation des droits ;
- la diffusion des évènements partagés ;
- la gestion des groupes hiérarchiques, c'est-à-dire gestion des E-blocs (abonnement, désabonnement, barrières d'entrée, de synchronisation et de sortie, et les informations sur les E-blocs.

A.5 Spécification du runtime

Le runtime MASL a été spécifié. La Figure 116 montre les principales classes du runtime. Il utilise des classes dérivées de `java.util.concurrent.CyclicBarrier` et de `java.util.concurrent.Future` pour la gestion des barrières de synchronisation et des labels.

Figure 116 Diagramme de classe UML des principales classes du runtime Java

A.6 Traduction MASL vers Java (ou autre langage embarqué)

Le processus de traduction MASL est indépendant du caractère réparti/centralisé de la plateforme. Le problème est déporté au niveau du runtime MASL. Aussi allons nous nous focaliser sur une implantation centralisée permettant un portage facilité vers une solution répartie.
On envisage un pré-processing pour ne garder que les instructions compatibles avec le type de l'agent et les instructions manipulant des variables de types autorisés par son langage cible. Cependant si des conversions sont acceptables, les types autorisés par le langage cible sont enrichis.
Pour la répartition entre traduction / appel au runtime, on retient le principe suivant : les tâches locales sont gérées par le processus de traduction tandis que tout ce qui consiste en des synchronisations (gestion de la coopération) entre agents, est plutôt gérée par le runtime qui repose sur la notion de partage.
La traduction de MASL vers le langage consiste à insérer dans le point d'entrée de l'exécutif de l'agent des appels au runtime MASL. Par exemple, pour un kherera télé opéré, le programme de contrôle est dans la méthode `void run()` d'une classe dérivant de `java.lang.Thread`. Pour un sbot, le point d'entrée est la méthode C `int MAIN()` par exemple.
La phase de préprocessing peut valider les fichiers XML de type à l'aide des XML Schemas. Elle peut vérifier que les méthodes de compensations ne concerne que pour les méthodes admettant une invocation non bloquante.
Elle génère la mission centrée sur l'agent en prenant en compte l'hétérogénéité des agents, leur sous-typage implicite.
Par exemple, voici quelques traitements effectués par cette phase :

- toute portion de code ayant une variable de type incompatible avec le langage cible est remplacée par des `.nop()` ;
- toute instruction faisant appel à un membre non présent dans le type de l'agent est remplacée par un appel à l'instruction `.nop()` ;
- lorsqu'il y a incompatibilité lors de l'initialisation de variables dans un E-bloc (type incompatible avec langage cible, valeur déterminée par un membre non présent dans le type), l'initialisation ne peut être faite par l'agent. Si c'est une variable partagée, elle sera faite par un agent ayant tout de compatible.
- lorsqu'une variable de type incompatible est présente dans un test d'un E-bloc, l'évaluation est toujours fausse ;
- lorsque dans un test de E-bloc, il y a évaluation d'une fonction ou d'un attribut non présent dans l'API de l'agent, l'évaluation est toujours fausse.
- Si une évaluation d'un test est toujours fausse pour cet agent, le code concerné ne sera pas inclus dans le programme de contrôle.

Soit le programme MASL suivant, conforme aux déclarations de la Figure 112 :

```
01| import MAAM.xml as MAAM;
02| import Clock.xml as Clock;
03| MAAM maam = newAgent(MAAM);
04| Clock clock = newAgent(Clock);

05| synchronous entry main (true) {
06|   shared int Variable1=10;
07|   local float Variable2=1.2f;
08|   shared event Event1;
09|   local event Event2;
10|   setAcceptState("normal");
11|   emitWithMessage(Event1, "message");
12|   Variable2=1;
13|   clock.logln("main:CA MARCHE !!!");

14|   asynchronous entry e1 (Variable1==10) {
15|     .logln("e1:CA MARCHE !!!");
16|     react (Event1) {
17|       .logln("Event1 in e1");
18|       restart;
19|     }
20|     react (Event2) {
21|       .logln("Event2 in e1");
22|       if (Variable2==1.2f) reelect(1) else reelect(2);
```

```
23|      }
24|    }
25|    react (Event1) {
26|        .logln("Event1 in main");
27|        resume;
28|    }
29|    react (Event2) {
30|        .logln("Event2 in main");
31|    }
32| }
```

Il peut se traduire en java de cette manière dans la classe de l'agent généré par le processus de traduction :

```
// Méthode donnant la valeur d'un test de bloc Entry dans la classe d'agent
public boolean isConform(Entry e) {
  try {
    if (e.getName().equals("main"))
      return true;
    if (e.getName().equals("e1"))
      return (((Integer)this.getVariable("Variable1")).intValue()==10);
    return false;
  } catch (Exception ex) {
    return false;
  }
}
// Différents sauts en fonction des blocs Entry après les évènements
public enum Return {
  NONE,
  MAIN_REELECT, MAIN_RESTART, MAIN_EXIT,
  E1_REELECT, E1_RESTART,E1_EXIT
}
// Politique de retour du dernier évènement et bloc Entry actuel
// Voir valeurs renvoyées par process
private Return policy(int process) {
  if (process==0)
    return Return.NONE;
  if (this.getActualEntry().getName().equals("main") && process==-1)
    return Return.MAIN_RESTART;
  if (this.getActualEntry().getName().equals("main") && process==-2)
    return Return.MAIN_EXIT;
  if (this.getActualEntry().getName().equals("main") && process>0)
    return Return.MAIN_REELECT;
  if (this.getActualEntry().getName().equals("e1") && process==-1)
    return Return.E1_RESTART;
  if (this.getActualEntry().getName().equals("e1") && process==-2)
    return Return.E1_EXIT;
  if (this.getActualEntry().getName().equals("e1") && process==1)
    return Return.E1_REELECT;
  if (this.getActualEntry().getName().equals("e1") && process>1)
    return Return.MAIN_REELECT;
  return Return.NONE;
}

// la méthode process renvoie la politique de retour suite au traitement
des évènements. Dans un bloc synchrone, la barrière de synchronisation est
appliquée dans cette méthode. Les appels externes vers l'agents sont
traités. La méthode process implante l'algorithme SDI. Les valeurs de
retour sont :
* 0=RESUME;       -1=EXIT;          -2=RESTART;
* 1=REELECT;       2=REELECT(2);    3=REELECT(3);

//La méthode run() contient le programme de contrôle.
```

```
public void run() {
  super.run();
  Return rt=Return.MAIN_REELECT;
  run_loop:
  while (true) {
    switch (rt) {
      case MAIN_REELECT:
        if (!this.isConform(Entry.getInstanceFromName("main"))) {
          rt=Return.MAIN_EXIT;
          break;
        }
        Entry.getInstanceFromName("main").registerAgent(this,
Entry.AgentInEntryState.WAITINGIN);
        try {
          Entry.getInstanceFromName("main").getWaitingInBarrier().await();
        } catch (Exception ex) {}
      case MAIN_RESTART:
        Entry.getInstanceFromName("main").registerAgent(this,
Entry.AgentInEntryState.RUNNING);
        this.setCurrentPermeability("normal");
        rt=policy(process());if (rt!=Return.NONE) break;
        Event e=new Event (this, "Event1","message");
        Event.broadcast( e);
        rt=policy(process());if (rt!=Return.NONE) break;
        try {
          this.setVariable("variable2",(float)1);
        } catch (Exception ex) {};
        rt=policy(process());if (rt!=Return.NONE) break;
        try{
          Object[] p = {"main:CA MARCHE !!!"};
          externalCall(clock, "logln", p);
        } catch (Exception ex) {};
        rt=policy(process());if (rt!=Return.NONE) break;
      case E1_REELECT:
        if (!this.isConform(Entry.getInstanceFromName("e1"))) {
          rt=Return.E1_EXIT;
          break;
        }
      case E1_RESTART:
        Entry.getInstanceFromName("e1").registerAgent(this,
Entry.AgentInEntryState.RUNNING);
        try{
          Object[] p = {"e1:CA MARCHE !!!"};
          internalCall("logln", p);
        } catch (Exception ex) {};
        rt=policy(process());if (rt!=Return.NONE) break;
      case E1_EXIT:
        Entry.getInstanceFromName("e1").registerAgent(this,
Entry.AgentInEntryState.OUT);
      case MAIN_EXIT:
        Entry.getInstanceFromName("main").registerAgent(this,
Entry.AgentInEntryState.WAITINGOUT);
        try {
          Entry.getInstanceFromName("main").getWaitingOutBarrier().await();
        } catch (Exception ex) {}
        Entry.getInstanceFromName("main").registerAgent(this,
Entry.AgentInEntryState.OUT);
        break run_loop;
    }
  }
}
```

```
// La méthode react contient les réactions aux évènements
public int react(Entry e) {
  if (e.getName().equals("main")) {
    if(this.cEvent.getName().equals("Event1")){
      //do something
      logln("Event1 in main");
      return 0; //RESUME
    }
    if(this.cEvent.getName().equals("Event2")){
      //do something
      logln("Event2 in main");
      //EXIT
    }
  }
  if (e.getName().equals("e1")) {
    if(this.cEvent.getName().equals("Event1")){
      //do something
      logln("Event1 in e1");
      return -2; //RESTART
    }
    if(this.cEvent.getName().equals("Event2")){
      //do something
      logln("Event2 in e1");
      try {
        if (((Float)this.getVariable("Variable2")).intValue()==1.2f)
          return 1; //REELECT(1)
        else
          return 2; //REELECT(2)
      } catch (Exception e1) {}
    }
  }
  return -1; //EXIT
}
```

Dans cette traduction, un E-bloc synchrone nécessite d'attendre avant d'entrer et de sortir :

```
Entry.getInstanceFromName("main").getWaitingInBarrier().await();
…
Entry.getInstanceFromName("main").getWaitingOutBarrier().await();
```

Une barrière de synchronisation est aussi appelée dans la méthode process() pour les agents dans un bloc synchrone.

Bien entendu, la classe nécessite le runtime MASL, donc les importations suivantes :

```
import valoria.masl.runtime.DataServer.VariableException;
import valoria.masl.runtime.OwnerTypeException;
import valoria.masl.runtime.Agent;
import valoria.masl.runtime.Entry;
import valoria.masl.runtime.Event;
import valoria.masl.runtime.util.MASLLog;
```

Bibliographie

Si la référence ne se trouve pas en bibliographie, elle se trouve en webographie.

[ADA, 83], ADA. *The Programming Language ADA Referenee Manual*. LNCS 155, Springer Verlag, 1983.

[Agha, 86] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, Series in Artificial Intelligence, MIT Press, 1986.

[Alami & all, 98] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. *An architecture for autonomy*. International Journal of Robotics Research, 17(4):315–337, apr 1998.

[Albus & all, 87] J. S. Albus & all. *NASA/NBS "Standard Reference Model for Telerobot Control System Architecture (NASREM)"*. NBS Technical Note 1235, National Bureau of Standards, Gaithersburg, MD, 1987.

[Arkin, 98] Arkin R., *Behavior-Based Robotics*, MIT Press, 1998.

[Arnaud, 00] Pierre Arnaud, *Des moutons et des robots, architecture de contrôle réactive et déplacement collectif des robots*, Presses Polytchniques et Universitaires Romandes, 2000.

[Alur & all, 00] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. *Modular specification of hybrid systems in CHARON*. In HSCC, pages 6-19, 2000

[Alur & all, 01] R. Alur & all, "*Hierarchical Hybrid Modeling of Embedded Systems*" Proceedings of EMSOFT'01: First Workshop on Embedded Software, October 8-10, 2001

[Amiguet, 00] M. Amiguet, *MOCA : Un modèle componentiel dynamique pour les systèmes multi-agents organisationnels*, thèse de doctorat, 2000

[Anitescu & all, 97] M. Anitescu and F. Potra, *Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementary problems*, ASME Nonlinear Dynamics 14, pp 231-247, 1997.

[Arai& all, 02] Tamio Arai, Enrico Pagello, Lynne E. Parker, *Editorial: Advances in Multi-Robot Systems*, Ieee Transactions On Robotics And Automation, Vol. 18, No. 5, October 2002: 655-661

[Armstrong, 97] J. Armstrong "*The development in Erlang*", ACM sighpla international Conference on Functional Programming p 196-203. 1997

[Atkin & all, 99] M. Atkin, D. Westbrook, and P. Cohen. "*HAC : a unified view of reactive and deliberative activity*. Notes of the European conf on artificial intelligence 1999

[Balch & all, 95] Balch, Arkin, *Communication in Reactive Multiagent Robotic Systems*, Autonomous Robots, 1995.

[Barraff, 93] David Baraff, *Issues in computing contact forces for non penetrating rigid bodies*, Algorithmica, pp 292-352, Oct. 1993.

[Baillie & all, 06] Jean-Christophe Baillie, Antoine Robin, *URBI,Tutorial for urbi 0.9* Basé sur URBI révision 142

[Baginski & all, 99] B. Baginski, A. Baumann, and S. Riesner. *Robcl - an distributed object oriented robot programming language*. In Proceedings of the 8th International Workshop on Robotics in Alpe-Adria-Danube Region (RAAD-99), pages 229–234, 1999.

[Bayindir & all, 07] Levent Bayindir and Erol Sahin, *A Review of Studies in Swarm Robotics*, Turk J Elec Engin, VOL.15, NO.2 2007.

[Bellifemine & all, 03] F. Bellifemine, G. Rimassa, and A. Poggi, *JADE – A FIPA-compliant agent framework* (in "4th International Conference on the Practical Applications of Agents i n s e a r c h o f i n n o v a t i o n exp - Volume 3 - n. 3 - September 2003 85 and Multi-Agent Systems (PAAM-99)"), The Practical Application Company Ltd., London, UK, 1999.

[Beaudry, 05] J. Beaudry, *Machine décisionnelle pour systèmes multi-robots coopératifs*, mémoire de maîtrise de génie mécanique, Université de montréal, 2005

[Benjamin & all, 04] D. Paul Benjamin & all "*Integrating perception, language an problem solving in a cognitive agent for mobile robot*" AAMAS'04 july 19-23 2004, New-York

[Benveniste & all, 02] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone, *The Synchronous Languages Twelve Years Later*, Proc. of the IEEE, special issue on Embedded Systems, 2002, to appear.

[Berry & all, 92] G. Berry and G. Gonthier. *The Esterel synchronous programming language: Design, semantics, implementation*. Science of Computer Programming, 19(2):87-152, 1992.

[Berry, 93] G. Berry, R. K. Shyamasundar, and S. Ramesh. *Communicating reactive processes*. In Proc. 20th A CM Conf. on Principles of Programming Languages, POPL '93, Charleston, Virginia, 1993.

[Blank & all, 99] Blank, D.S., Hudson, J.H., Mashburn, B.C., Roberts, E.A. (1999). *The XRCL Project: The University of Arkansas*' Entry into the AAAI 1999 Mobile Robot Competition. Technical Report CSCE-1999-01.

[Blank & all, 06] Douglas Blank, Deepak Kumar, Lisa Meeden, and Holly Yanco, *The Pyro toolkit for AI and robotics*, AAAI Spring Symposium, Stanford, CA , ETATS-UNIS (22/03/2004) 2006, vol. 27, no 1 (95 p.) pp. 39-50, 2006

[Brooks, 91] R. A. Brooks. *Intelligence without representation*. Artificial Intelligence, 47:139–159, 1991.

[Bougé, 96] Luc Bougé, *The Data-Parallel Programming Model: a Semantic Perspective*, INRIA Research Report RR-3044, Nov. 1996.

[Boniol 95] F. Boniol. *Synchronous communicating reactive processes*. In Models and Proofs, 2nd AMAST Workshop on Real-Time Systems, Bordeaux, France, June 14-16 1995.

[Borrelly & all, 98] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. *The orccad architecture*. The Int. Journal of Robotics Research, 17(4) :338–359, April 1998.

[Boussinot, 91] F.Boussinot *Reactive-C: An extension of C to Program Reactive Systems*. Software Practice and Experience, 21(4):401-428, 1991.

[Boussinot & all, 95] F. Boussinot, G. Doumenc and J.B. Stefani, *Reactive Objects*, INRIA Research Report RR-2664, Oct. 1995.

[Boussinot & all, 98] F. Boussinot, L. Hazard, J-F Susini. *Distributed Reactive Machines*, Proc RTCSA'98, Hiroshima, IEEE, 1998.

[Bratman, 87] Bratman, M. E. [1987] (1999). *Intention, Plans, and Practical Reason*. CSLI Publications. ISBN 1-57586-192-5.

[Brenner, 03] M. Brenner. *A Multiagent Planning Language*. In Workshop on PDDL, ICAPS'03, Trento, Italy, 2003.

[Brooks, 91] Brooks R., *Intelligence without Reason*, Proceedings of the IJCAI'91, Sydney (Australie), Morgan-Kaufmann, pp. 569-595, 1991.

[Browning & all, 03] B. Browning and E. Tryzelaar, *Übersim: A Multi-Robot Simulator for Robot Soccer*, Autonomous Agents and Multi-Agent Systems (AAMAS'03), under submission.

[Cao & all, 97] Cao Y. Cao, A. Fukunaga, A. Kahng, "*Cooperative Mobile Robotics: Antecedents and Directions*", Autonomous Robots, vol. 4, pp. 7-23, 1997.

[Carpenter & all, 98] Carpenter B, Zhang G, Fox G, Li X, Wen Y. *HPJava: data parallel extensions to Java*. Concurrency: Practice and Experience 1998; 10(11–13):873–877.

[Caspi, 93] P. Caspi. *Lucid synchrone*. In International Workshop on Principles of Parallel Computing (OPOPAC), November 1993.

[Caspi & all, 96] Paul Caspi and Marc Pouzet. *Synchronous Kahn Networks*. In ACM SIGPLAN International Conference on Functional Programming, Philadelphia, Pensylvania, May 1996.

[Chauvin, 82] Rémy Chauvin, *Les sociétés animales*, Quadrige / Presses Universitaires de France, Septembre 1982.

[Chien & all, 00] Chien, S. ; Knight, R. ; Stechert, A. ; Sherwood, R. ; and Rabideau, G. 2000. Using *iterative repair to improve the responsiveness of planning and scheduling*. In Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS).

[Coe 95] Fabien Coelho. *Compilation of I/O Communications for HPF*. In 5th Symposium on the Frontiers of Massively Parallel Computation, pages 102-109, February 1995.

[Cottle & all, 68] R. W. Cottle and G. B. Dantzig. *Complementary pivot theory of mathematical programming*. Linear Algebra and its Applications, 1:103-125, 1968.

[Dai & all, 02] X. Dai, G. Hager, and J. Peterson. *Specifying behavior in c++*. In IEEE Int. Conf. on Robotics and Automation, ICRA'02, volume 1, pages 153–160, Washington DC, USA, May 2002.

[Dallaway & all, 93] Dallaway JL, Mahoney RM, Jackson RD (1993) *CURL - a robot control environment for Microsoft Windows*. Proceedings of the RESNA 93 Annual Conference. 510-511

[Dastani & all, 03] M. Dastani & L. van der Torre "*Programming Boid-Plan agents deliberating about conflicts along defeasible mental attitudes and plans*" AAMAS 2003

[Dias & all, 06] M. Dias, R. Zlot, N. Kalra, and A. Stentz, "*Market-based multirobot coordination : a survey and analysis*," Proc. of the IEEE, vol. 94, no. 7, pp. 1257–1270, 2006.

[Dorigo & all, 04] Marco Dorigo, Elio Tuci, Roderich Groß, Vito Trianni, Thomas Halva Labella, Shervin Nouyan, Christos Ampatzis, Jean-Louis Deneubourg, Gianluca Baldassarre, Stefano Nolfi, Francesco", *The SWARM-BOT Project*", url = "citeseer.ist.psu.edu/dorigo04swarmbot.html"

[Dorigo & all, 04b] M. Dorigo, E. Sahin, *"Swarm Robotics - Special Issue*", Autonomous Robots, vol. 17, pp. 111-113, 2004.

[Drogoul & all, 98] Drogoul A. et A. Collinot (1998). "*Using the Cassiopeia Method to Design a Soccer Robot Team*." Applied Articial Intelligence (AAI) Journal 12(2-3): 127-147.

[Dubois & all, 03] M. Dubois, Y. Le Guyadec and D. Duhaut, "*Control of Interconnected Homogeneous Atoms: Language and Simulator*". Proc. of the 6th Int. Conf. on Climbing and Walking Robots and the Support Technologies for Mobile Machines (CLAWAR), pp 391-398, 2003

[Dubois & all, 08] M. Dubois, Y. Le Guyadec and D. Duhaut, "*MASL: a Language for Multi-Agent System*". Chapter in the book "Multiagent Systems", Robotics & Automation Open Access Platform, ars-journal.com, Vienna, Austria, EU, 2008 (to appear)

[Dudek & all, 93] G. Dudek, E. Jenkin, D. Wilkes, "*A taxonomy for swarm robots*", In Proc. 1993 IEEE International Conference on Intelligent Robots and Systems, pp 441–447, 1993.

[Duhaut, 02] D.Duhaut, *Robotic Atom* Clawar 2002, 24-26 september 2002 Paris (France)

[Duhaut & all, 06] Dominique Duhaut, Claude Gueganno, Yann Le Guyadec, Michel Dubois, *Horocol language and Hardware modules for robots*, 1st National Workshop on "Control Architectures of Robots: software approaches and issues", April 6-7, 2006 Montpellier - FRANCE2006.

[Duhaut & all, 06b] D. Duhaut, C. Gueganno, Y. Le Guyadec, M. Dubois, *Tools for Building a team of robots* Robotics and Automation Conference RAC'06, 13-14 Mars 2006 Cebu, Philippines

[Duhaut & all, 08] Dominique Duhaut, Yann Le Guyadec, Michel Dubois, *Programming a multi-agent system with MASL*, *IEEE/ASME AIM 2008*, July 1-5, 2008 Chine 2008.

[Dylla & all, 03] F. Dylla, A. Ferrein, and G. Lakemeyer. *Specifying multirobot coordination in ICPGolog – from simulation towards real robots*. In Proc. of the Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World Modeling, Planning, Learning, and Communicating (IJCAI 03), 2003.

[Dudek & all, 96] Dudek, Jenkin, Milios, Wilkes, *A Taxonomy for Multi-Agent Robotics*, Autonomous Robots, 1996.

[Dutta & all, 04] Dutta, I. Bogobowicz, A.D. Gu, J.J., *Collective robotics - a survey of control and communication techniques*, Intelligent Mechatronics and Automation, 2004. Proceedings. 2004 International Conference on, pages 505- 510, August 2004

[Endo & all, 04] Yoichiro Endo, Douglas C. MacKenzie and Ronald C. Arkin, Fellow, Usability Evaluation of High-Level *User Assistance for Robot Mission Specification*, IEEE Transactions On Systems, Man, And Cybernetics—Part C: Applications And Reviews, Vol. 34, No. 2, May 2004.

[Elliott & all, 97] C. Elliott and P. Hudak. *Functional reactive animation*. In International Conference on Functional Programming, pages 263–273, June 1997.

[Farinelli & all, 04] A. Farinelli, L. Iocchi, and D. Nardi, "*Multirobot systems: A classification focused on coordination*," IEEE Trans. Syst., Man, Cybern. B, vol. 34, no. 5, pp. 2015. http://citeseer.ist.psu.edu/farinelli04multirobot.html

[Ferber, 87] J. Ferber, *Des Objets aux Agents: une Architecture Stratifiée*, 6ième Congrés Afcet de Reconnaissance des Formes et Intelligence Artificielle (RFIA'87), Vol. 1, pages 275-286, Dunod, Novembre 1987.

[Ferber, 95] Jacques Ferber, *Les systèmes multi-agents : Vers une intelligence collective*, InterEditions, 1995

[Ferber & all, 00] Jacques Ferber et Olivier Gutknecht, *Operational semantics of Multi-agent Organizations*, LCNS Intelligent Agents VI, LNAI 1757, pp 205-217, 2000.

[Ferber & all, 05] Ferber J., Michel F., Baez J., *AGRE : Integrating Environments with Organizations*,in E4MAS'04: Environments for Multiagent Systems, Melbourne, Australie, p. 127-134, 2005.

[Firby, 89] James Firby, *Adaptive Execution in Complex Dynamic Worlds*, PhD thesis, Yale University, 1989.

[Finin & all, 97] T. Finin and Y. Labrou. *KQML as an agent communication language*. . J.M. In: Bradshaw (ed.), Software Agents, pp. 291-316. Cambridge, MA, 1997.

[Flynn, 72] Flynn, M., *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972

[Fortune & all, 78] S. Fortune and J.Wyllie. *Parallelism in RandomAccess Machines*. In Proceedings of the 10th Annual Symposium on Theory of Computing, pages 114–118, 1978.

[Gamcet, 05] J. Gamcet, *Systèmes multi-robots aériens : architecture pour la planification, la supervision et la coopération*, thèse de doctorat, 2005.

[Geist & all, 94] A. Geist, A. Beguelin, J. Dongarra,W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA, USA: MIT Press, 1994.

[Ghafoor & all, 04] Abdul Ghafoor, Mujahid ur Rehman, Zaheer Abbas Khan, H. Farooq Ahmad, Arshad Ali, Hiroki Suguri, "*SAGE: Next Generation Multi-Agent System*", pp. 139-145, Vol. 1. PDPTA, Navada (USA), June 2004.

[Gat, 97] E. Gat. *ESL: a language for supporting robust plan execution in embedded autonomous agents.* In Proceedings of the IEEE Aerospace Conference, 1997.

[Gat, 98] E. Gat. *On three-layer architectures*. In Artificial Intelligence and Mobile Robots. AAAI Press, 1998.

[Gueganno & all, 04] Gueganno C., Duhaut D., *A hardware/software architecture for self reconfigurable robots*, Proceedings of the 7th international Symposium on Distributed Autonomous Robotics Systems (DARS 04).

[Gueganno & all, 05] Claude Guéganno and Dominique Duhaut, "*Remote Tools using Wireless Communication for Self-Reconfigurable Robot*, Applied to MAAM Project",in proceedings of IEEE International Symposium on Computational Intelligence in Robotics and Automation, 2005. Espoo (Finland)

[Gueganno & all, 05b] Claude Guéganno and Dominique Duhaut, "*A Versatile and Open Software/Hardware Architecture for Mechatronic Systems*" in proceedings of IEEE International Symposium on Industrial Electronics 2005, June 20-23, Dubrovnik (Croatia).

[Giavitto & all, 01] J.L. Giavitto and O. Michel, *MGS: a Programming Language for the Transformations of Topological Collections*, LaMI technical report N° 61, May 2001.

[Gibson, 86] James Jerome Gibson, *The Ecological Approach to Visual Perception, L'approche écologique de la perception visuelle*, Gibson, J.J. (1979). The Ecological Approach to Visual Perception. Boston: Houghton Mifflin.ISBN 0898599598 (1986)

[Gutknecht, 01] Gutknecht, O. *Proposition d'un modèle organisationnel générique de systèmes multiagent et examen de ses conséquences formelles, implémentatoires et méthodologiques*. Thèse d'informatique, Montpellier II, 2001.

[Gutknecht & all, 01] Gutknecht O., Michel F., Ferber J., « *Integrating tools and infrastructure for generic multi-agent systems* », in 5th Int. Conf. on Autonomous Agents, Montréal, 2001.

[Halbwachs & all, 91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "*The synchronous data flow programming language LUSTRE*," Proceedings of the IEEE, vol. 79, pp. 1305–1320, Sept. 1991.

[Halbwachs 98] HALBWACHS N., « *Synchronous Programming of Reactive Systems – A Tutorial and Commented Bibliography* », Actes de CAV'98, no 1427 LNCS, Springer, p. 1–16.

[Halbwachs & all, 02] HALBWACHS N., BAGHDADI S., « *Synchronous Modelling of Asynchronous Systems*», Actes de EMSOFT 2002, no 2491 LNCS, Springer, p. 240–251.

[Halstead, 85] HALSTEAD, JR., R. H. *Multilisp: A language for concurrent symbolic computation*. ACM Trans. Program. Lang. Syst. 7, 4 (Oct. 1985), 501-538.

[Harrouet & all, 02] Harrouet, F., Tisseau, J., Reignier, P. and Chevaillier, P., *oRis : un environnement de simulation interactive multi-agents*, Revue des Sciences et Technologies de l'Information, série Technique et Science Informatiques (RSTI-TSI), 21(4) :499-524, Oct 2002.

[Hempel, 94] R. Hempel, "*The MPI standard for message passing*," in HPCN Europe 1994: Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II, (London, UK), pp. 247–252, Springer-Verlag, 1994.

[Hewitt, 77] C.E. Hewitt, *Viewing Control Structures as Patterns of Passing Messages*, Journal of Artificial Intelligence, Vol. 8 No 3, pages 323-364, 1977.

[Howden & all, 01] Nick Howden, Ralph Rönnquist, Andrew Hodgson, Andrew Lucas, *JACK Intelligent Agents Summary of an Agent Infrastructure*, 5th International Conference on Autonomous Agents, 2001

[Horswill, 00] I. Horswill. *Functional programming of behavior-based systems*. Autonomous Robots, 9(1):83–93, 2000.

[Hudak & all, 02] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson "*Arrows, robots, and functional reactive programming*" LNCS 159-187 Spinger Verlag 2002

[Hughes, 00] John Hughes. *Generalising monads to arrows*. Science of Computer Programming, 37:67{111, May 2000.

[Huntsberger & all, 03] T.L. Huntsberger, P. Pirjanian , A. Trebi-Ollennu, H. Das, H. Aghazarian, A.J. Ganino, M.S. Garrett, S.S. Joshi, and P.S. Schenker. *CAMPOUT: A Control Architecture for Tightly Coupled Coordination for Multi-robot Systems for Planetary Surface Exploration*. IEEE Transactions on Systems, Man, and Cybernetics, Volume 33, Number 5, 2003, pp.555-559

[Gazi & all, 06] V. Gazi, B. Fidan, "*Coordination and Control of Multi-agent Dynamic Systems: Models and Approaches*", In E. S¸ahin, W. Spears and A. Winfield, editors, Proceedings of the Second International Workshop on Swarm Robotics at SAB 2006, volume 4433 of Lecture Notes in Computer Science, pages 71-102. Springer Verlag, Berlin, Germany, 2006

[Ghallab & all, 94] Malik Ghallab and Hervé Laruelle, "*Representation and Control in IxTeT, a Temporal Planner*", In Proceedings AIPS-94, pp61-67, Chicago, 1994

[Gomaa, 00] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications With UML*, Addison Wisley, 2000.

[Ingrand & all, 92] F. Ingrand, M. Georgeff, A. Rao, « *An architecture for real-time reasoning and system control* », IEEE Expert, 7(6):34-44, 1992.

[Ingrand & all, 96] F. F. Ingrand & all "*PRS : a high level supervision and control language for autonomous mobile robots*", IEEE int cong on robotics and automation Minneapolis, 1996

[Ingrand & all, 02] F. Ingrand, F. Py, *An Execution Control System for Autonomous Robot*. In IEEE International Conference on Robotics and Automation, 2002

[Ingrand, 05] F. Ingrand, *Architectures Logicielles pour la Robotique Autonome*, JNRR 03 Journées Nationales de Rechercher en Robotique, October 8-10, 2003, Murol/Clermont-Ferrand, France.

[Iocchi & all, 01] L. Iocchi, D. Nardi, and M. Salerno, *Reactivity and Deliberation: a survey on Multi-Robot Systems* in Balancing Reactivity and Deliberation in Multi-Agent Systems (LNAI 2103), E. P. M. Hannebauer, J. Wendler, Ed. Springer, 2001, pp. 9--32. http://citeseer.ist.psu.edu/iocchi01reactivity.htm

[Jakobi, 98] N. Jakobi, *The minimal simulation approach to evolutionary robotics*, In Gomi, T., editor, Evolutionary Robotics - From Intelligent Robots to Artificial Life (ER'98). AAI Books, Ontario, Canada.

[Jorgensen & all, 04] M.W. Jorgensen, E.H. Ostergaard, H. Hautop, "*Modular ATRON: Modules for a self reconfigurable robot*" in proceedings of 2004 IEE/RSJ International conference on Intelligent Robots an Systems (IROS 2004).

[Joyeux, 07] Sylvain Joyeux, *Un composant logiciel pour la gestion et l'exécution de plan en robotique : Application aux systèmes multi-robots*, Thèse de doctorat Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS/CNRS), Décembre 2007.

[Jung, 99] Christoph G. Jung, *Theory and Practice of Hybrid Agents*, PHD thesis, 1999.

[Kaebling & all, 98] Kaebling L. P., Littman M. L., Cassandra A. R., *Planning and Acting in Partially Observable Stochastic Domains*, Artificial Intelligence, vol. 101, num. 1–2, pp. 99-134, 1998.

[Kauppi, 03] Kauppi, Ilkka, *Intermediate Language for Mobile Robots: A Link between the High-Level Planner and Low-Level Services in Robots*, VTT Technical Research Centre of Finland, Dissertation for the degree of Doctor of Science in Technology, 2003

[Kitano & all, 97] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, « *RoboCup : The Robot World Cup Initiative* », Proceedings of the 1st International Conference on Autonomous Agents, Johnson, Hayes-Roth eds, 1997, p. 340-347, ACM Press.

[King, 02] G. King "*Tapir : the evolution of an agent control language*" American association of artificial intelligence 2002

[Kirsh, 08] Alexandra Kirsch. *Integration of Programming and Learning in a Control Language for Autonomous Robots Performing Everyday Activities*. Ph.D. Thesis, Technische Universität München, 2008.

[Klavins, 03] E. Klavins "*A formal model of a multi-robot control and communication task*" IEEE Conf on Decision and Control, 2003

[Klavins, 04] E. Klavins "*A language for modeling and programming cooperative control systems*" Int Conf on Robotics and Automation ICRA 2004

[Klein, 02] Klein, J. 2002. *Breve: a 3D simulation environment for the simulation of decentralized systems and artificial life*. Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems. The MIT Press. http://www.spiderland.org/breve.pdf

[Konolige, 97] Kurt Konolige. *Colbert : A language for reactive control in sapphira*. Technical report, SRI International, June 1997.

[Kubika & all, 01] Jeremy Kubica, Arancha Casal, Tad Hogg: *Complex Behaviors From Local Rules In Modular Self-Reconfigurable Robots*. ICRA 2001: 360-367

[Labrou & all, 99] Y. Labrou, T. Finin, and Y. Peng, *Agent Communication Languages: The Current Landscape*, IEEE Intelligent Systems, vol. 14, no. 2, pp. 45-52, 1999.

[Lamping & all, 97] Yim, M.; Lamping, J.; Mao, E.; Chase, J. G. *Rhombic dodecahedron shape for self - assembling robots*. SPL Technical Report P9710277. Palo Alto CA: Xerox PARC; 1997.

[Laue & all, 05] Tim Laue, Kai Spiess et Thomas Röfer : *SimRobot — a general physical robot simulator and its application in RoboCup*. In RoboCup 2005 : Robot Soccer World Cup IX, Lecture Notes in Artificial Intelligence, 2005.

[Le Guernic & all, 91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "*Programming real-time applications with SIGNAL*," Proceedings of the IEEE, vol. 79, pp. 1321–1336, Sept. 1991.

[Le Guyadec & all, 05] Y. Le Guyadec, C. Guégano, M. Dubois, D. Duhaut, "*Using HoRoCoL to program a society of agents or teams of robot*", The 6th IEEE Symposium on Computational Intelligence in Robtics and Automation, Helsinki University of Technology, 27 - 30 June 2005, Finland.

[Le Guyadec & all, 05b] "*Using HoRoCoL to Control Robotics Atoms*", Y. Le Guyadec, C. Guégano, M. Dubois, D. Duhaut, IEEE International Conference on Mechatronics and Automation, July 30 - August 1 2005, Niagara Falls, Ontario, Canada.

[Levesque & all, 97] Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. *Golog: A logic programming language for dynamic domains*. Journal of Logic Programming.

[Lorenz, 84] Konrad Lorenz, *Les Fondements de l'éthologie* Champs Flammarion 1984, Paris

[Lozano-Perez & all, 86] T. Lozano-Perez & R. Brooks "*An approach to automatic robot programming*" Proceedings of the 1986 ACM fourteeth annual conf on computer science 1986, ACM Press

[Lötzsch, 04] M. Lötzsch. *XABSL - A Behavior Engineering System for Autonomous Agents*. Diploma thesis. Humboldt-Universität zu Berlin, 2004.

[Lucidarme & all, 02] P. Lucidarme, O. Simonin, & A. Liégeois, *Implementation and Evaluation of a Satisfaction/Altruism Based Architecture for Multi-Robot Systems*, In Proceedings of the 2002 IEEE International Conference on Robotics & Automation, pages 1007-1012, Washington, DC, 2002.

[Lundh & all, 04] R. Lundh, L. Karlsson & A. Saffiotti. *Dynamic Configuration of a Team of Robots*. In Proc. of the European Conference on Artificial Intelligence (ECAI'04) Workshop on Agents in Dynamic and Real-Time Environments, 2004.

[McDermott, 93] McDermott, Drew (1993). *A reactive plan language*. Technical report, Yale University, Computer Science Dept.

[MacKenzie & all, 97] D. C. MacKenzie, R. C. Arkin, and J. M. Cameron, "*Multiagent mission specification and execution*," Auton. Robot., vol. 4, no. 1, pp. 29–52, 1997.

[Malone, 90] T. W. Malone. *Organizing information processing systems: parallels between human organizations and computer systems*. In W. W. Zachary and S. P. Robertson, editors, Cognition, Computation and Cooperation, pages 56-83. Ablex, 1990.

[Makatchev, 00] *Human-Robot Interface Using Agents Communicating* In An XML-Based Markup Language." By Maxim Makatchev (Department of Manufacturing Engineering and Engineering Management, City University of Hong Kong) and S. K. Tso (Centre for Intelligent Design Automation and Manufacturing, City University of Hong Kong). Pages 270-275 (with 25 references) in Proceedings the Ninth IEEE International Workshop on Robot and Human Interactive Communication [IEEE RO-MAN 2000, Osaka, Japan, September 27-29, 2000]. Piscataway, NJ, USA: IEEE Computer Society, 2000.

[Malenfant & all, 03] http://www.univ-ubs.fr/valoria/Jacques.Malenfant/ARM, J. Malenfant and S. Denier, *ARM : un modèle réflexif asynchrone pour les objets répartis et réactifs*, Actes de LMO 2003, Revue L'Objet, Hermès/Lavoisier, vol. 9, no 1-2, pp. 91-103, 2003.

[Mataric & all, 96] M. J. Mataric and D. Cliff, "*Challenges in evolving controllers for physical robots*", Journal of Robotics and Autonomous Systems, vol. 19, pp. 67–83, Oct. 1996.

[Mataric, 97] Maja J. Mataric. *Behavior-based control: Examples from navigation, learning, and group behavior*. Journal of Experimental & Theoretical Artificial Intelligence, 1997.

[Matura & all, 02] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, S. Koraji, "*M-TRAN: Self-Reconfigurable Modular Robotic System*" in IEEE/ASME transactions on mechatronics, Vol.7 No.4 2002

[Miglino & all, 95] Orazio Miglino, Henrik Hautop Lund, Stefano Nolfi: *Evolving Mobile Robots in Simulated and Real Environments*. Artificial Life 2(4): 417-434 (1995)

[Minar & all, 96] Minar, Nelson, Burkhart, Rogert, Langton, Chris, & Askenazi, Manor. 1996. *The Swarm Simulation System : A Toolkit for Building Multi-Agent Simulations*. Santa Fe Institute Working Paper #96-06-042.

[Mintzberg, 79] H. Mintzberg. *The Structuring of Organizations*. Englewoods Cliffs, 1979.

[Mirtich, 98] Brian Mirtich, *V-Clip: Fast and Robust Polyhedral Collision Detection*, ACM Transactions on Graphics, Vol 17, No 3, pp 177-208, July 1998.

[Munoz, 03] Munoz, A. *Coopération située : une approche constructiviste de la conception de colonies de robots*, Thèse de doctorat, 2003

[Muscettola, 02] Muscettola N., Dorais G. A., Fry C., Levinson, R. and Plaunt C. 2002. *Idea : Planning at the core of autonomous reactive agents*. In Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space.

[Michel, 96] O. Michel. *Design and implementation of 8 1/2, a declarative data-parallel language*. special issue on Parallel Logic Programming in Computer Languages, 1996

[Mondada & all, 03] F. Mondada & all "*Swarm–bot : for concept to implementation*", IEEE/RSJ int conf on intelligent robots and systems IROS 2003

[Nana & all, 03] Laurent Nana, Jerôme Legrand, Frank Singhoff et Lionel Marce, *Modélisation, simulation et test des algorithms d'interpretation de plans PILOT*, 4e Conférence Francophone de MOdélisation et SIMulation "Organisation et Conduite d'Activités dans l'Industrie et les Services" MOSIM'03 – du 23 au 25 avril 2003 - Toulouse (France)

[Nair & all, 03] R. Nair, M. Tambe & S. Marsella. *Role Allocation and Reallocation in Multiagent Teams : Toward a Practical Analysis*. In Proceedings of the second International Joint Conference on Agents and Multiagent Systems (AAMAS), 2003.

[Nembrini & all, 07] Julien Nembrini & Alcherio Martinoli, *Robotique en Essaim, Récents Résultats et Directions Futures*, JNRR 07.

[Nishiyama et al., 1998] Nishiyama, H., Ohwada, H. and Mizoguchi, F. [1998]. *A Multiagent Robot Language for Communication and Concurrency Control*. Proceedings of the International Conference on Multiagent Systems, 3–7 July 1998. Pp. 206–213.

[Odell, 98] Odell J., *Advanced Object-Oriented Analysis and Design using UML*, Cambridge University Press, 1998.

[Odell, 02] Odell J., *Objects and Agents Compared*. Journal of Object Technology, 1(1):41–53, 2002.

[Odell & all, 05] Bernhard Bauer and James Odell, "*UML 2.0 and agents: how to build agent-based systems with the new UML standard*" Journal of Engineering Applications of Artificial Intelligence Volume 18, Issue 2 , March 2005, Pages 141-157.

[Parker, 98] Lynn Parker, *ALLIANCE: An Architecture for Fault Tolerant Multi-Robot Cooperation*, IEEE Transactions on Robotics, 1998.

[Parker, 00] L. E. Parker, "*Lifelong adaptation in heterogeneous multi-robot teams: Response to continual variation in individual robot performance*, " Autonomous Robots, vol. 8, no. 3, pp. 239--267, 2000. http://citeseer.ist.psu.edu/parker00lifelong.html

[Pell & all, 98] Barney Pell, Gregory A. Dorais, Christian Plaunt, Richard Washington, *The Remote Agent Executive: Capabilities to Support Integrated Robotic Agents*, Procs. of the AAAI Spring Symp. on Integrated Robotic Architectures, 1998.

[Pettinaro & all, 03] Pettinaro G., Kwee I., Gambardella L., Definition*, Implementation, and Calibration of the Swarmbot3D Simulator*, Technical Report No. IDSIA-21-03, December 16, 2003.

[Pembeci & all, 01] I. Pembeci & G. Hager "*A comparative review of robot programming languages*" report CIRL – Johns Hopkins University august 14, 2001

[Peterson & all, 99] J. Peterson, G.D. Hager, and P. Hudak. "*A language for declarative robotic programming*" Int Conf on Robotics and Automation ICRA 1999

[Pokahr & all, 05] Pokahr, A, Braubach, L and Lamersdorf, W, 2005, *Jadex: a BDI reasoning engine*. In Bordini, RH, Dastani, M, Dix, J and Fallah-Seghrouchni, AE (eds.), Multi-Agent Programming: Languages, Platforms and Applications (Multiagent Systems, Artificial Societies, and Simulated Organizations, 15), Berlin: Springer, ch. 6, pp. 149-174.

[Resnick, 94] Resnick, Mitchel, *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds*, MIT Press.

[Ricordel, 01] Ricordel, Pierre-Michel, *Programmation Orientée Multi-Agents : Développement et Déploiement de Systèmes Multi Agents Voyelles*, thèse de doctorat, 2001.

[Ricordel & all, 02] Ricordel, Pierre-Michel, & Demazeau, Yves. 2002. *Volcano, a Vowels- Oriented Multi-agent Platform*. Pages 253–262 of : Revised Papers from the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems. Springer-Verlag.

[de Rivera & all, 05] de Rivera, G.G.; Ribalda, R.; Colas, J. & Garrido, J., *A generic software platform for controlling collaborative robotic system using XML-RPC*, Advanced Intelligent Mechatronics. Proceedings, 2005 IEEE/ASME International Conference on, Volume , Issue , 24-28 July 2005, pp. 1336 – 1341

[Roux & all, 92] O. Roux, D. Creusot, F. Cassez and J.P. Elloy. *Le langage réactif asynchrone ELECTRE*. Technique et Science Informatique (TSI), 11(5):35-66, 1992

[Ross & all, 05] Ross, R, Collier, R and O'Hare G, 2005, *AF-APL: Bridging Principles and Practices in Agent Oriented Languages* (Lecture Notes in Computer Science, 3346). New York: Springer, pp. 66-88.

[Rubenstein & all, 04] M. Rubenstein, K. Payne, W-M. Shen, "*Docking among independent and autonomous CONRO self-reconfigurable robot*" in ICRA 2004

[Rus & all, 02] Daniela Rus, Zack Butler, Keith Kotay, Marsette Vona - *Self-reconfiguring robots* Communications of the ACM 45(3):39-45, 2002

[Russel & all, 06] Stuart Russell, Peter Norvig, *Intelligence artificielle*, 2ème edition, Pearson Education France, Paris, 2006

[Seq, 03] Y. Secq, *RIO: Rôles, Interactions et Organisations : une méthodologie pour les systèmes multi-agents ouverts*, Thèse de doctorat, 2003.

[Seuling & all, 06] Seugling, A., Rolin., M. 2006. *Evaluation of physics engines and implementation of a physics module ina 3d-authoring tool*. In Master's Thesis, Umea University Department of Computing Science. http://www.cs.umu.se/education/examina/Rapporter/SeuglingRolin.pdf

[da Silva & all, 02] da Silva, Joao Luis T., & Demazeau, Yves. 2002. *Vowels co-ordination model*. Pages 1129–1136 of : Proceedings of the first international joint conference on Autonomous agents and multiagent systems AAMAS'02. ACM Press.

[Savall, 03] M. Savall, *Une architecture d'agents pour la simulation : le modèle YAMAM et sa plate-forme Phoenix*, Thèse de doctorat, 2003

[Simmons & all, 98] R. Simmons and D. Apfelbaum, "*A Task Description Language for Robot Control*," Proceedings Conference on Intelligent Robotics and Systems, October, 1998.

[Simmons & all, 02] Simmons, Reid, Smith, Trey, Dias, M. Bernardine, Goldberg, Dani, Hershberger, David, Stentz, Anthony, and Zlot, Robert (2002), *A Layered Architecture for Coordination of Mobile Robots*, In Multi-Robot Systems: From Swarms to Intelligent Automata, Proceedings from the 2002 NRL Workshop on Multi-Robot Systems, Kluwer Academic Publishers.

[Simonin & all, 02] Olivier Simonin, Fabien Michel, Jérôme Chapelle, Jacques Ferber, *Un simulateur de systèmes multi robots dans MADKIT*, Lille France, JFIADSMA 2002.

[SDL 2000] Union Internationale des Télécommunications – Télécommunication Section Normalisation, Recommendation Z.100: « *Specification and Description Language* », Electronic Bookshop, Geneva, 2000.

[Scerri & all, 04] Paul Scerri, David V. Pynadath, Nathan Schurr, Alessandro Farinelli, Sudeep Gandhe and Milind Tambe, *Team Oriented Programming and Proxy Agents: The Next Generation*, Proceedings of 1st international workshop on Programming Multiagent Systems, 2004.

[Shoham, 91] Y. Shoham. « *AGENT-0: a simple agent language and its interpreter* » In Proceedings of the Ninth National Conference on Artificial Intelligence, Vol II (pp. 704 -709), Anaheim, CA, MIT Press, 1991.

[Shoham, 93] Y. Shoham, « *Agent Oriented Programming* », Artificial Intelligence, 60(1), 1993, p. 51-92, North-Holland.

[Smith, 80] R. G. Smith. *The contract net protocol: High-level communication and control in a distributed problem solver*. In IEEE Transaction on Computers, number 12 in C-29, pages 1104–1113, 1980.

[Stewart & all, 96] D. Stewart and J. Trinkle, *An implicit Time-Stepping Scheme for Rigid Body Dynamics with inelastic collisions and Coulomb Friction*, International Journal of Numerical Methods in Engineering, 1996

[Susini & all, 98] J.F. Susini, L. Hazard, F. Boussinot, *Distributed Reactive Machines*, INRIA Research Report RR-3376, March 1998.

[Sunderam, 90] V. S. Sunderam. *PVM: a framework for parallel distributed computing*. Concurrency, Practice and Experience, 2(4):315–340, 1990.

[Tambe & all, 99] Milind Tambe, W Shen, M Mataric, D Goldberg, Pragnesh J. Modi, Z Qiu, B Salemi, *Teamwork in cyberspace: Using TEAMCORE to make agents team-ready*, AAAI Spring Symposium on Agents in Cyberspace, 1999.

[Tidhar, 93] Tidhar, G. (1993) '*Team Oriented Programming: Preliminary Report*,' Technical Report 37, Australian Artificial Intelligence Institute.

[Tissue & all, 04] Tisue, S., & Wilensky, U. (2004). *NetLogo: Design and Implementation of a Multi-Agent Modeling Environment*. http://ccl.northwestern.edu/papers/agent2004.pdf

[Ugur, & all, 06] Ugur E., Dogar M., Soysal O., Cakmak M., Sahin E., *MACSim: Physics-based Simulation of the KURT3D Robot Platform for Studying Affordances*, D1.2.1 Public Deliverables, EU projects MACS.

[Verhaeghe & all, 05] Filip Verhaeghe, Stefaan Decorte, Didier Tytgadt, *Behavior based multi-agent systems as data types*, Patent WO/2005/069130, 2005

[Valiant, 90] L. G. Valiant. *A Bridging Model for Parallel Computation*. Communications of the Association for Computing Machinery, 33(8):103–11, August 1990.

[Vidal & all, 02] José M Vidal and Paul Buhler. *A generic agent architecture for multiagent systems*. Technical report, University of South Carolina, 2001. USC CSCE TR-2002-011.

[Volpe & all, 00] R Volpe, I Nesnas, T Estlin, D Mutz, R Petras, and H. Das. *CLARAty: Coupled layer architecture for robotic autonomy*. Technical report, NASA Jet Propulsion Laboratory, 2000.

[Wan & all, 00] Zhanyong Wan and Paul Hudak. *Functional reactive programming from first principles*. In Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI), pages 242{252, Vancouver, BC, Canada, June 2000. ACM, ACM Press.

[Weerasooriya & all, 95] D. Weerasooriya, A. S. Rao, and K. Ramamohanarao. *Design of a concurrent agent-oriented language*. In Intelligent Agents: Theories, Architectures, and Languages. Lecture Notes in Artificial Intelligence LNAI 890, Amsterdam, Netherlands, 1995. Springer Verlag.

[Weiss, 99] G. Weiss, '*Multiagent Systems: A modern approach to distributed artificial intelligence*', MIT Press, London, England, 1999.

[Waydo & all, 03], Stephen Waydo, Eric Klavins, *Specification of Control Tasks in CCL*:The Computation and Control Language, (Poster), SMC-IT Conference 2003, July, 2003.

[Wilkins & all, 95] David E Wilkins, Karen L Myers, John D Lowrance, and Leonard P Wesley, *Planning and reacting in uncertain dynamic environments*, Journal of Experimental an Theoretical AI, 7, 1995

[Wooldridge & all, 95] Wooldridge, M., Jennings, N.R.: *Intelligent agents: Theory and practice*. The Knowledge Engineering Review 10 (1995) 115—152

[Wooldridge & all, 00] M.Woodridge, N.R.Jennings and D.Kinny, "*The Gaia Mehodology for Agent-Oriented Analysis and Design*", Autonomous Agents&Multi-Agent Systems, Volume 3, Issue 3, September, 2000.

[Yonezawa & all, 87] A. Yonezawa, E. Shibayama, T. Takada et Y. Honda, *Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1*, dans [OOCP 87], pages 55-89.

[Yoshimura & all, 00] Kenichi Yoshimura, Ralph Rönnquist and Liz Sonenberg, *An Approach to Specifying Coordinated Agent Behaviour*, Lecture Notes In Computer Science; Vol. 1881; Proceedings of the Third Pacific Rim International Workshop on Multi-Agents: Design and Applications of Intelligent Agents, 2000

[Zielinski, 00] C. Zielinski "*Programming and control of multi-robot systems*" Conf. On Control and Automation Robotics and Vision ICRARCV'2000 dec 5-8 2000, Singapore

[Zhang & all, 98] T. Zhang, S. Covaci, Magedanz, T., *OMG and FIPA Standardisation for Agent Technology: Competition or Convergence?*, in Proceedings of the European ACTS AgentWorkshop, Brussels, February 1998.

Webographie

[Collier, 02] Collier, Nick. 2002. *RePast : the REcursive Porous Agent Toolkit*. http ://repast.sourceforge.net/.

[FIPA, 01] FIPA '*Communicative Act Library Specification'*, Rapport technique XC00037H, 15 Août 2001, http://www.fipa.org/specs/fipa00037/.

[FIPA, 02] FIPA, '*FIPA ACL Message Structure Specification* SC00061G', 6 Décembre 2002, http://www.fipa.org/specs/fipa00061/index.html.

[Gazebo, 03] Environnement de simulation 3D base ODE compatible Player/Stage : http://playerstage.sourceforge.net/gazebo/gazebo.html

[Havok, 02] http://www.havok.com/

[JADE, 00] JADE Java Agent DEvelopment Framework http://sharon.cselt.it/projects/jade/

[Java3D, 00] https://java3d.dev.java.net/

[JNI, 97] Java Native Interface : http://java.sun.com/javase/6/docs/technotes/guides/jni/

[Khepera, 02] API java publique des khéperas (k-team) : http://hem.passagen.se/foggy/khepera/khepera/Khepera.html

[Klein, 05] Klein J., *Breve: a 3d Simulation Environment for Multi-Agent Simulations and Artificial Life*. http://www.spiderland.org/breve/index.php

[KODEX, 05] Kovan ODE Extension : http://kovan.ceng.metu.edu.tr/software/macsim/kodex-0.6.4.tar.gz

[Kurt3D, 06] API C++ publique du robot Kurt3D (robot mobile de type Kurt2 avec laser 3D) : http://kovan.ceng.metu.edu.tr/software/macsim/macsim-1.0.0.tar.gz

[lpzrobots, 06] projet lpzrobos de l'université de Leipzig à base de ODE : http://robot.informatik.uni-leipzig.de/software

[MAAM, 05] API publique C++ du composant robotique MAAM : http://www-valoria.univ-ubs.fr/Michel.Dubois/index_API2.htm

[Madkit, 01] http://www.madkit.org/

[MissionLab, 97] http://www.cc.gatech.edu/aimosaic/robot-lab/research/MissionLab

[Microsoft Robotic Studio, 07] http://msdn.microsoft.com/fr-fr/robotics/default(en-us).aspx

[Mpi, 89] http://www-unix.mcs.anl.gov/mpi/

[NetLogo, 00] The NetLogo system. http://ccl.northwestern.edu/netlogo/.

[NovodeX, 02] NovodeX. Middle-ware Physics Software Provider. NovodeX Physics SDK v2, http://www.novodex.com/ (voir maintenant [physX, 06]).

[ODE, 02] ODE. Opensource Project, Multibody Dynamics Software. Open Dynamics Engine. http://opende.sourceforge.net

[ODEJava, 04] OdeJava, *Open Dynamics Engine binding for Java*, http://odejava.org/OdejavaIntro/

[physX, 06] http://www.nvidia.com/object/nvidia_physx.html

[Polypod, 97] http://robotics.stanford.edu/users/mark/polypod.html

[Polybod, 98] http://www2.parc.com/spl/projects/modrobots/polybot/polybot.html

[Pvm, 06] http://www.csm.ornl.gov/pvm/

[RePast, 03] *RePast - Recursive Porus Agent Simulation Toolkit*. http://repast.sourceforge.net/index.html

[Reynols, 86] http://www.red3d.com/cwr/boids/

[Simulatom, 04] Simulateur du projet MAAM : http://www-valoria.univ-ubs.fr/Michel.Dubois/index_sim.htm

[Simulator-bob, 03] Simulateur de robots mobiles base sur ODE : http://simbob.sourceforge.net/

[Simrobot, 05] Simulateur 3d cinématique de l'université de bremen basé sur ODE :
http://www.informatik.uni-bremen.de/simrobot/index_e.htm

[S-bots, 05] API publique C++ des S-bots :
http://kovan.ceng.metu.edu.tr/software/KODEX-old/swarmbot3d-2.0.0.tar.gz

[Smith, 99] MathEngine Non technical presentation slides http://ode.org/slides/slides.html

[StarLogo, 00] The StarLogo system. http://education.mit.edu/starlogo/.

[Swarmbot, 02] http://www.swarm-bots.org

[SWIG, 95] Simplified Wrapper and Interface Generator : http://www.swig.org

[Trinkle, 02] J. Trinkle, Trinkle's Survey of Multibody Dynamics Software:
http://www.cs.rpi.edu/~trink/sim_packages.html

[Ubersim, 02] http://www.cs.cmu.edu/~robosoccer/ubersim/

[UML 2 Superstructure Specification, 07] http://www.omg.org/docs/formal/07-11-02.pdf

[Vortex, 02] CM-labs Vortex http://www.cm-labs.com/

[XODE, 04] XODE http://tanksoftware.com/xode/

Résumé

L'approche classique des langages pour le contrôle de Systèmes Multi-Agents (SMA), a fortiori robotiques et autonomes, consiste d'abord en un point de vue microscopique : chaque agent dispose de son propre programme de contrôle contenant des primitives de communication / synchronisation permettant la coopération / collaboration entre agents. L'émergence d'un comportement global, le point de vue macroscopique du calcul, ne peut qu'être observé a posteriori.

Dans ce contexte, MASL propose une approche unifiée et macroscopique à l'expression de calculs hétérogènes et distribués sur des agents conçus en suivant le modèle délibératif, réactif ou hybride. C'est un langage de haut niveau indépendant de l'exécutif où chaque agent, vu comme une entité concurrente, détermine localement sa participation à des blocs d'exécution collectifs (e-blocs). Chaque e-bloc est un programme collectif anonyme pouvant s'exécuter sur un réseau d'agents selon des critères locaux. Le mode d'orchestration (scalaire, synchrone, asynchrone) est déterminé statiquement par un attribut du bloc, les communications supportent le modèle à mémoire partagée, le modèle à envoi de messages et le modèle d'évènements. L'hétérogénéité des agents est assurée par héritage et polymorphisme alors que l'autonomie est proposée par un mécanisme (appelé perméabilité) de filtrage où chaque agent peut masquer/ouvrir son interface dynamiquement et selon la position de l'émetteur dans la hiérarchie d'e-blocs. Dans un contexte d'allocation dynamique des agents, de reprise après échec ou de remplacement d'un agent robotique dans une flotte de robots (cas d'une panne ou perte de fonctionnalité compromettant la mission), le e-bloc propose une perspective de point d'entrée d'un traitement collectif. Dans le cas d'e-bloc synchrones, le paradigme sous-jacent est issu du modèle data-parallèle, permettant ici des traitements itératifs par vagues successives d'agents. Au final, MASL propose des avancées dans le domaine des SMA (appartenance dynamique à des groupes, précision du rythme des actions à entreprendre pour permettre une coopération désirée) et au niveau de la gestion des erreurs.

Mots clés : Système Multi Agents, robotique collective, langage de contrôle, parallélisme, langage synchrone, Système auto reconfigurable.

Abstract

The classical approach for Multi-Agent System (MAS) Control, especially autonomous and robotic ones, deals first from a microscopic point of view: each agent embed a control program with communication/synchronization primitives that enable cooperation between agents. The emergence of a global behaviour from a macroscopic point of view can only be observed afterwards.

In this context, MASL offers a macroscopic and unified approach with heterogeneous and distributed calculations over deliberative, reactive or hybrid agents. In this high level language, regardless of the runtime, each concurrent agent locally decides its participation in a collective execution block named an e-block. Each e-block is an anonymous collective program that runs over an agent network following local conditions.

The orchestral mode (scalar, asynchronous, synchronous) is statically fixed by a shared block attribute. The communication use shared memory, events, synchronous messages passing, and asynchronous messages passing. Heterogeneous agents are managed with heritage and polymorphism. Permeability mechanism, dealing with agent autonomy, allows an agent to dynamically filter calls to its interface in respects to the sender position in the e-block hierarchy.

In dynamic task allocation of agents, auto failover and recovery, agent replacement in a robot fleet (case of agent failure, loss of a mandatory functionality for the mission) an e-block is an entry point of a collaborative work. In the case of synchronous e-block, the programming paradigm is the data parallel model with iterative task for waves of agents.

Finally, MASL offers advances in the field of MAS (dynamic belonging to groups, accuracy of the pace of actions to undertake to enable a desired cooperation) and for the management of errors.

Keywords : Multi-Agents System, Collective Robotics, Control Language, Parallelism, Synchronous Language, Modular Self-Reconfigurable Robotic System.

UBS
UNIVERSITÉ DE BRETAGNE-SUD

N° d'ordre : 133

Université de Bretagne-Sud

BP 92116 - 56321 LORIENT CEDEX

Tél : +33(0) 2 97 87 66 66 Fax : +33(0) 2 97 87 66 00