MASL, langage de contrôle multi-agents robotiques

Michel Dubois, Yann Le Guyadec et Dominique Duhaut

Mots clés : Système Multi-Agents, robotique collective, langage de contrôle, parallélisme, langage synchrone, Système auto reconfigurable.

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 (Multi-Agent System Language) 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 ou bloc entry). 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.

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. Pour répondre à cette difficulté, nous avons défini 6 propriétés qu'un langage d'agents doit posséder. Un modèle de programmation est ensuite proposé. Il sera décrit dans la section 1. La section 2 s'intéresse aux travaux précédants du domaine des multi-agents pour la robotique ainsi qu'aux fondements de notre modèle. La section 3 propose un état de l'art des architectures robotiques que doit prendre en compte MASL. La section 4 donne la sémantique informelle. La section 5 propose une illustration par l'exemple dans le contexte de la RoboCup [Kitano & all, 97]. La section 6 décrit l'implantation actuelle. La dernière section conclut sur les avancées que propose MASL ainsi que sur les perspectives.

1. Notre modèle de programmation

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.

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. La section 6 décrit plus précisément l'implémentation.

2. Les travaux liés

On peut dresser un résumé (*Cf.* le Tableau 1) de la couverture des propriétés précédentes concernant les langages de contrôle spécifiquement orientés agents :

- GOLOG [Levesque & all, 97] (alGOl in LOGic) ;
- MRL [Nishiyama & all, 98] (Multiagent Robot Language) ;
- Le langage d'actions TAPIR [King, 02] ;
- CDL [MacKenzie & all, 97] (Configuration Description Language) ;
- XABSL [Lötzsch, 04] (eXtensible Agent Behavior Specification Language) ;
- HoRoCoL [Dubois & all, 03], [Duhaut & all, 06], [Duhaut & all, 06b], [Le Guyadec & all, 05], [Le Guyadec & all, 05b] (Homogeneous Robotic Component Language).

De plus les languages multi-agents hybrides 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) portent sur des domaines de valeurs discrets et des domaines de valeurs continues.

Tableau 1 Langages orientés agents robotiques et les six propriétés (les numéros correspondent à la numérotation des propriétés de la section précédante)

	(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).

Mais MASL s'inspire aussi et retient :

- des langages data-parallèles comme HPJava [Carpenter & all, 98], les méthodes de synchronisme d'exécution qui serviront à l'implémentation de MASL ;
- des langages acteurs comme ABCL/1 [Yonezawa & all, 87] (Actor Based Concurrent Language), les messages asynchrones et l'interface dynamique qui introduiront la notion de perméabilité en MASL ;
- des langages réactifs comme dans le modèle DROM [Boussinot & all, 98] (Distributed Reactive Object Model), le synchronisme et l'asynchronisme de réaction au changement de l'environnement et le broadcast d'événements que MASL couvrira avec les « `events` » ;

- 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 ».

3. Les architectures des agents

MASL doit pouvoir implémenter des architectures d'agents robotiques. Aussi, il convient de les étudier.
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.
Les architectures d'agents robotiques se caractérisent par une mise en compétition des modules pour remplir la mission. 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 1 montre les trois types d'architectures robotiques logicielles.

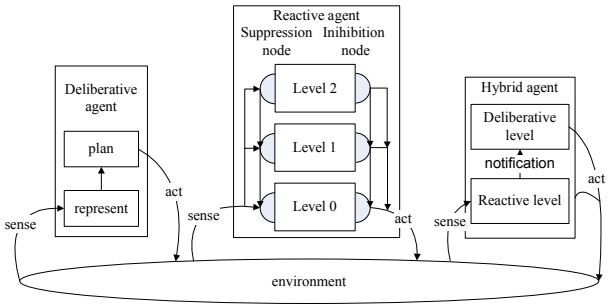

Figure 1 : Les agents mettent en œuvre des modèles d'exécution qui implémentent les concepts délibératifs, réactifs ou hybrides.

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.
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 sévèrement des problèmes.
L'arbitrage entre précision et temps de calcul est critique voire impossible à trouver du fait de la complexité des algorithmes de manipulation symbolique. 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. 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.

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 (en cas d'absence d'apprentissage).
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. 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.
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.

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, notamment pour la gestion des erreurs.
L'architecture hybride permet de dépasser les restrictions des langages comportementaux qui utilisent des méthodes de type réactif pour la programmation de robots. Elle propose des langages intermédiaires dans lesquels on va spécifier la façon d'exécuter le plan. Ils sont aussi appelés langages de supervision, de planification réactive. Notamment, PRS [Ingrand 96], ESL [Gat 97] et TDL [Simmons 98] sont des exécutifs robustes appartenant à cette catégorie. 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'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.

4. Semantique informelle

Dans cette section, le terme agent est utilisé pour indiquer le programme qui fait appel aux primitives du robot.

4.1 Agents hétérogènes

L'objectif est de programmer un ensemble d'agents hétérogènes pour qu'ils travaillent ensemble. Ensuite, l'approche proposée consiste à importer la liste des capacités des agents à partir d'une description XML.

Exemple:

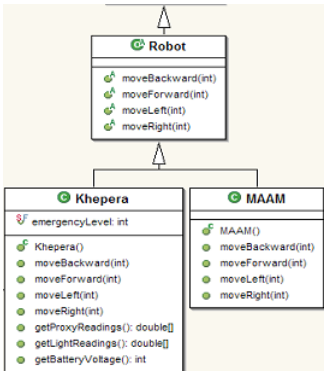

Figure 2: XML description of the robot primitives

Un fichier XML décrit le robot Khepera et un composant robotique MAAM [Duhaut, 02] (Molecule = Atom | Atom + Molecule). Un ensemble de primitives est défini pour chacun d'eux.

```
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);
```

Les lignes 01, 02 définissent un type MASL à l'aide de la description contenue dans le fichier XML. Ensuite, les lignes 03, 04 sont des instanciations de 5 agents K1 et K2 du type Khepera et M1, M2, M3 de type MAAM.

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

Le bloc `main` est exécuté par les 5 agents. La sémantique de `.moveLeft(30);` est une exécution par l'agent de l'instruction qui correspond à ligne java `this.moveLeft(30);`. Chaque robot exécute son propre code de façon autonome dans ce premier exemple. Pour l'instant, il n'est pas possible de prédire l'ordre d'exécution de ces instructions par les 5 agents.

4.2 Parallelisme synchrone et asynchrone

L'exemple précédent est une exécution asynchrone dans laquelle tous les agents exécutent leur code de façon indépendante.

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

Ici la différence est la spécification synchrone dans le bloc principal. La spécification synchrone signifie que, après l'exécution de chaque instruction, tous les agents à l'intérieur du e-bloc (ici tous les agents parce que le test est toujours vrai) attendront la fin de l'exécution de tous les autres.

Dans ce cas, les mouvements de tous les robots sont effectués simultanément. Une fois de plus, il n'est pas possible de prédire exactement l'ordre d'exécution, car un agent peut prendre plus de temps dans l'accomplissement de son action, auquel cas toutes les autres seront en attente.

La notion d'e-bloc est également utilisée pour définir une section de code qui sera exécutée par un sous-ensemble des agents. Par exemple

```
05| asynchronous entry example(.isMAAM()) {
06|    .moveLeft(30);
07|    .moveForward(10);
09| }
```

définit une entrée d'instruction (e-bloc) nommée `example` avec un test (`.isMAAM()`). L'évaluation de ce test (`.isMAAM()`) choisit l'agent autorisé à exécuter la séquence de la ligne 06, 07 et 08. L'agent qui ne satisfait pas le test se déplace vers la prochaine instruction (ligne 10). L'instruction est utilisée pour former des groupes d'agents. Il est possible de décrire le comportement de ce sous-ensemble : se déplacer à gauche puis aller en avant.

MASL propose aussi une entrée scalaire

```
01| scalar entry e1(test)
```

pour définir un e-bloc dans lequel un seul agent est autorisé à entrer. Le premier agent répondant au critère `test` entre dans l'entrée et la verrouille afin que les autres la saute.
La notion d'entrée (e-bloc) peut être rapprochée à l'instruction accept du langage Ada[ADA, 83].

4.3 Communication à l'aide de variables et d'évènements

La portée des variables ou des événements dépend de l'endroit où ils sont définis et des modificateurs `local/shared` employés.

```
01| asynchronous entry main (true) {
02|   shared int sglobalvar=0;
03|   synchronous entry e1 (.isMAAM()) {
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|   }
11| }
```

Dans cet exemple, tous les agents entrant dans main se partagent la variable déclarée à la ligne 02 `sglobalvar`. Cela signifie que seul un `sglobalvar` existe dans le run-time et de toute modification par tout agent change sa valeur.
L'entrée à la ligne 03 sélectionne un sous-ensemble de robot (`.isMAAM()`).Ainsi, la variable `svar` définie à la ligne 04 est partagée uniquement pour les 3 agents dans cette partie du code.
La variable `lvar` définie à la ligne 05 (donc 3 `lvar` différentes sont définies : une par agent) est instanciée localement par chaque agent qui se trouve dans cette partie de code.
En ligne 06, chaque agent incrémente sa variable interne `lvar`. Par conséquent, la valeur finale écrite en ligne 08 sera 1 (3 fois si il y a 3 agents de type MAAM). Mais la valeur finale `svar` est 3 parce que chacun des 3 agents MAAM va incrémenter sa valeur. Notez que cette valeur est écrite 3 fois à cause de la programmation synchrone de l'entrée `e1`.

Selon le même principe, il est possible de définir un événement. Un événement peut être global à tous les agents s'il est défini dans le bloc `main` ou restreint à un groupe s'il est défini dans une autre entrée. Un événement `local` est visible uniquement par l'agent qui l'a émis. La seule instruction est **`emit`** `(event)`.

```
01| asynchronous entry main (true) {
02|   shared event sevent;
03|   asynchronous entry e1 (.MAAM()) {
04|       …
06|     react (sevent) {
07|       .log("Reaction to an event");
08|     }
```

A la ligne 02, il y a la déclaration globale de l'événement nommé `sevent` dans l'entrée `main`. L'agent entrant dans l'entrée `e1` exécute le code (ligne 04). Si, au cours de son exécution, le `sevent` est émis par un agent dans une autre section du code, alors l'exécution normale stoppe et il y a saut à la section `react` qui définit les réactions aux évènements à la manière d'un `catch` en Java. L'événement `sevent` est recherché dans la liste et le code correspondant sera exécuté.
Si le code (ligne 08) inclut l'instruction MASL **`resume`** (reprendre) alors l'agent retourne à l'endroit où il était dans le code (ligne 04) quand l'événement a été émis. Cette construction permet à un agent de répondre à un événement et le retour aux travaux en cours.

4.4 Integration dynamique d'agent dans un groupe

Le problème est de permettre à un agent de quitter un groupe pour rejoindre un autre. Pour quitter le groupe, nous pouvons terminer l'exécution normale d'une entrée en utilisant l'instruction **`break`** ou utiliser l'instruction **`reelect`** (réélire) qui revient à la dernière entrée et vérifie à nouveau le test. Par exemple, si dans l'exemple précédent à la ligne 08 se trouve **`reelect`** alors l'agent se confronte à nouveau au test de l'entrée `e1` à la ligne 03. Cela revient à tester si l'agent est toujours un robot MAAM ou non. Si oui, il entre de nouveau dans l'e-bloc `e1`, ou il va à la prochaine instruction (ligne 09).
Cette construction est utile pour extraire un agent d'un groupe. Toutefois, le problème est alors d'ajouter l'agent à un autre groupe. Il faut donc imaginer qu'il trouve une autre entrée dont le test est vrai.
Enfin nous avons également défini des instructions pour verrouiller ou déverrouiller une entrée. Cela permet à certains agents de contrôler le nombre d'agents entrant dans un e-bloc.

4.5 Envoi synchrone/asynchrone de messages

Dans la définition XML du Khepera, la primitive `moveLeft` est définie. Elle peut être utilisée de deux manières. Tout d'abord, un agent k1 veut aller à gauche, son code est alors les suivantes:

```
01 : .moveLeft(30);
```

exprimant que l'instruction est appliquée à l'agent k1 par lui-même.

Ou le code de l'agent k1 contient :

```
01 : k2.moveLeft(30);
```

alors k1 demande à l'agent k2 de se déplacer vers la gauche. Dans ce cas, on peut imaginer deux scénarios :

- l'exécution de k1 est bloquée jusqu'à ce que l'exécution de la primitive de déplacement vers la gauche de k2 termine ;
- l'exécution de k1 peut continuer pendant l'exécution de la primitive qui fait se déplacer vers la gauche k2.

Cela dépend de la définition de la primitive `.moveLeft()`.
Le fichier XML définissant les services du robot peut décrire deux types de primitives en fonction de la valeur du champ synchronous_call. Quand il est vrai, k1 est bloqué et on a un envoi synchrone de message.
Pour illustrer l'envoi asynchrone de message, on introduit une fonction retournant un entier comme `getBatteryVoltage (): int` dans le fichier XML de Khepera. Ensuite, le code de k1 pourrait être :

```
02: li = k2.getBatteryVoltage ();
```

exprimant qu'une variable locale `li` de k1 reçoit la valeur de la tension de la batterie de k2. Dans le cas d'un appel asynchrone, k1 poursuit son travail. Si, à un certain endroit dans le code, k1 veut utiliser la valeur `li`, il doit être sûr que l'assignation de `li` est accomplie.
Pour résoudre ce problème, nous introduisons le type **`Label`**, inspiré de l'interface java `java.util.concurrent.Future`.

```
01 : Label llabel;
02 : llabel.li= k2.getBatteryVoltage ();
…
06 : if (isFinished(llabel)){…}
```

La ligne 01 déclare un nouveau label `llabel`. Ligne 02 fait un appel asynchrone d'une primitive et l'attache au label `llabel`. Il est alors possible à la ligne 06 de tester si les instructions qui lui sont attachées sont terminées ou non. Notez qu'il est possible d'attacher plus d'une instruction d'un agent à un label, ainsi que des instructions différentes venant de différents agents. L'instruction **`isFinished`** permet d'éviter le blocage de l'agent tandis que l'utilisation d'une variable qui prend une valeur future provoque le blocage de l'agent jusqu'à son assignation.

4.6 Perméabilité dynamique

La notion de perméabilité est connectée à la notion précédente d'envoi de message. Elle est utilisée pour exprimer que certains agents pourraient ne pas être en mesure de répondre à un appel d'une primitive à un certain moment pendant l'exécution.
La perméabilité est définie dans le fichier XML du robot. Il définit un ensemble d'états du robot et la liste des primitives qui peuvent être exécutées dans chacun de ces États.
Par exemple, un état de perméabilité `standard` détermine s'il est possible pour un agent MAAM d'exécuter toutes les 4 primitives défini dans la figure 1. Mais un deuxième état de perméabilité `damage`, seule la primitive `moveForward (int)` pourrait être appelée. Cet état correspond à un robot ayant des problèmes.
En outre, pour tous les états de la perméabilité, les primitives sont protégés pour l'exécution en vertu de ces niveaux: global, groupe et local. Cela signifie que selon l'état de la perméabilité courant, une primitive peut être exécutée par tous

les agents dans le e-bloc `main`, que par les membres du groupe (ceux dans la même entrée que l'agent destinataire du message) ou uniquement par l'agent lui-même.
L'état de perméabilité peut être dynamiquement modifié que par l'agent lui-même par l'exécution d'une instruction spécifique **`setAcceptState`**`(string)`, où `string` est le nom de l'état de perméabilité.
La langage MASL fournit également une instruction **`wait`**`(string)`. Cette instruction est utilisée pour mettre un agent en mode d'attente jusqu'à son activation par un appel de l'une des primitives visibles dans l'état de perméabilité définis par `string`. Durant cette attente, il se comporte comme un agent passif tout de même contraint par son état de perméabilté actuel, contrairement à un objet passif qui a une interface statique.

5. L'expressivité de MASL dans une application de la RoboCup

Comme exemple du langage MASL, nous proposons de distinguer trois groupes de robots: l'un est un attaquant et le second est un défenseur et un est l'entraîneur.

```
05| asynchronous entry main (true) {
06|   shared event mvBack, mvForward, move;
07|   scalar entry coach (true) {
08|     local  int lv, li=0;
09|     loop
10|       lv=.analyseSituation(); li++;
11|       if (lv<0) emit mvBack;
          else emit mvForward;
12|       if (li==100) {li=0; emit move;};
13|     endloop
14|   }
15|   asynchronous entry attack(.isFast()) {
16|     loop .playAttack(); endloop;
17|     react(mvForward)
        {.moveForward(20);resume;};
18|     react(mvBack)
        {.moveBackward(20);resume;};
19|   }
20|   asynchronous entry defense (true) {
21|     loop .playDefense(); endloop;
22|     react(mvForward)
        {.moveForward(5);resume;};
23|     react(mvBack)
        {.moveBackward(5);resume;};
24|     react(move)
        {.setRandomFast();reelect(2);};
25|   }
26| }
```

Cet exemple est construit pour montrer certaines caractéristiques :
- la définition des groupes de robots ;
- l'extensibilité pour le passage à l'échelle ;
- le changement dynamique de groupe ;
- la façon de contrôler un groupe par un superviseur.

L'instanciation des agents n'est pas représentée ici. Nous supposons qu'il s'agit d'un ensemble d'agents pour des robots identiques. Tous ces agents se partageront 3 événements (ligne 6).
Pendant l'exécution, le premier agent à exécuter la ligne 7 devient l'entraîneur (parce que c'est un **`entry`** scalaire, un seul ne peut entrer). Les autres agents à arriver à la ligne 7 vont passer à l'instruction ligne 15. Si le test effectué sur eux-mêmes (`.isFast()`) est vraie, alors ils entrent et vont à la ligne 16 pour jouer en mode attaque, les autres se déplacent à la ligne 20 où ils entreront (parce que le test est vrai) et exécutent la ligne 21 pour jouer en mode défense.
Nous pouvons voir ici que le premier groupe d'agents est séparé en 3 groupes: un (seul) dans l'entrée coach, un ensemble d'agents dans l'ebloc `attack` et le reste dans le bloc `defense`. Notez que ceci est indépendant du nombre de robots.
L'évolution dynamique est illustrée au travers du comportement de l'entraîneur. Il analyse la situation (ligne 10) et décide des actions des deux groupes, les attaquants et les défenseurs. Ainsi, à la ligne 11, il émet l'évènement `mvBack` (resp. `mvForward`). Cet événement est global à tous les agents et ils peuvent y réagir.
Les attaquants vont réagir (ligne 17 (resp. 18) par un `moveForward (20)` (resp. `moveBackward (20)`), puis revenir à leur comportement offensif à la ligne 16 lors de l'exécution de **`resume`**.

Les défenseurs vont réagir (ligne 22 (resp. 23) par un `moveForward (5)` (resp. `moveBackward (5)`) et ensuite revenir à leur ligne de conduite naturelle de défense à la ligne 21.
Nous pouvons aussi voir ici comment un groupe est invité à faire de grandes amplitudes (`20`), qui caractérisent l'attaque, tandis que l'autre se contente de faire « des pas »(`5`).
L'entraîneur peut aussi forcer les défenseurs à devenir des attaquants. Toutes les 100 boucles (ligne 12), l'entraîneur émet un événement `move`. Dans ce cas, seuls les défenseurs vont réagir (ligne 24). Leur réaction est décider au hasard s'ils sont rapides ou non. Après quoi, ils vont rentrer dans la ligne 5 du programme par l'exécution de l'instruction **`reelect`**`(2)`. Ils vont entrer à nouveau dans le bloc `main`. Parce que l'entrée `coach` est verrouillé, ils se déplaceront à la ligne 15 pour tester s'ils deviennent des attaquants (si oui, ils entrent dans la ligne 16) ou non (ils se déplacent à la ligne 20) ou deviennent des défenseurs de nouveau.
Bien entendu, l'exemple est simpliste et on peut le rendre plus réaliste en raffinant les comportements.

6. Implémentation

Nous allons décrire les implémentations envisagées de MASL dans des contextes robotiques divers. On se focalise dans un premier temps sur la chaine de production des programmes de contrôle et de la traduction d'un programme MASL vers une plateforme cible. Puis nous envisagons les scénarii de déploiements.

6.1 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. 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 3a), le traducteur utilise la définition de type XML et le programme MASL pour générer la boucle réactive/délibérative. En utilisant JavaCC [JavaCC, 97], PMD [PMD, 04] (utilitaire d'analyse statique de code), le traducteur d'un code source MASL vers un code java s'appuit sur le runtime MASL java.
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 [Java3D, 00] / ODEJava [ODEJava, 04], nommé SimExecBots, 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 3b), 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 [JNI, 97]). 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.

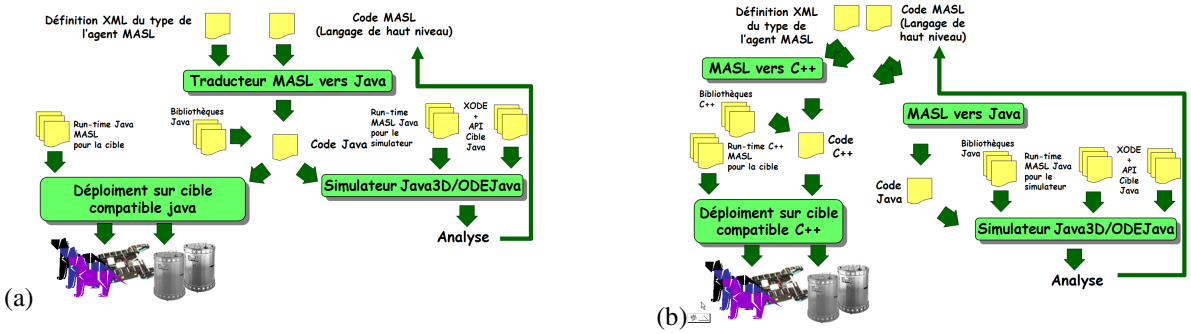

Figure 3 Principe de la traduction d'un programme de contrôle MASL vers une cible compatible (a) Java ou (b) C++

6.2 Scénarios de déploiement

Plusieurs scénarios de déploiement ainsi que des modèles d'exécution montrés en Figure 5 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 5, 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 fichiers XML des agents 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 termes 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.

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.

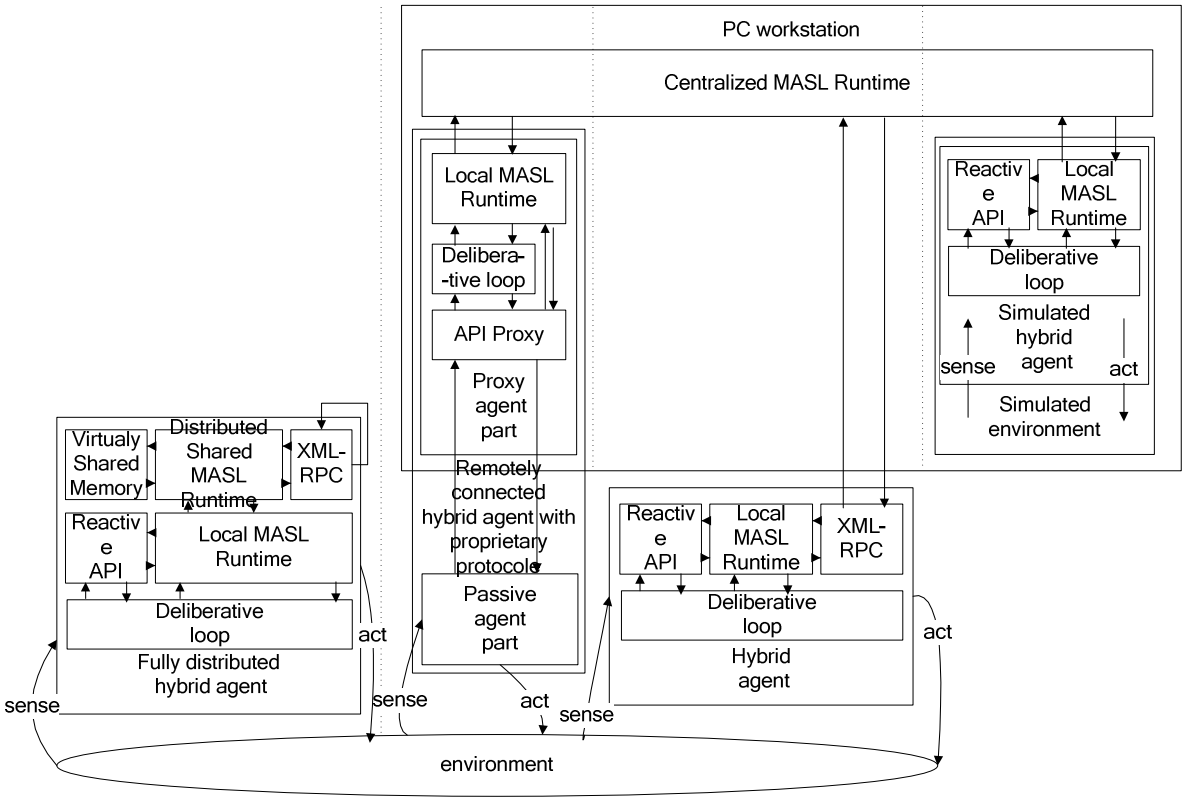

Figure 4 Scénarii de déploiement

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 eblocs (abonnement, désabonnement, barrières d'entrée, de synchronisation et de sortie, et les informations sur les eblocs.

7. Avancées et perspectives

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

7.1 Avancées

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 blocs **`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 réflexive. MASL permet de ce point de vue une généralisation de la notion de coopération au sein d'un groupe puisque il ne réduit

pas la communication de deux agents, comme AGR [Ferber & all, 00] 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 `synchronous` **`entry`** sont des outils puissants pour ce type d'implémentation.

Du point de vue de la sûreté 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 procédures spécifiques seront automatiquement lancées pour arrêter 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 propagation 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.

La Figure 5 à la page suivante représente la carte des fonctionnalités de MASL.

7.2 Perspectives

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 contentés 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 [Wooldridge & all, 00], Voyelles [Ricordel, 01]).

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 réutilisation, leur portabilité et leur introspection. La définition de MASL en π-ADL [Oquendo, 04] (Architecture Description Language) qui repose notamment sur les langages d'architecture (ADL) et le π-Calcul [Milner, 99], permettra d'appliquer ce dernier aux systèmes de la robotiques collective via un modèle de composition de

services distants et connectés entre eux dynamiquement pour réaliser une mission. L'architecture logicielle est devenue un thème scientifique majeur de l'informatique : elle fournit l'abstraction qui permet de développer rigoureusement et de faire évoluer des systèmes logiciels complexes au vu des besoins tant fonctionnels que non fonctionnels. Cette approche supporte la description (par leur spécification qui décrit leur comportement) et l'exécution (par composition) de services distants tout en contrôlant la qualité de cette exécution, en s'appuyant sur leur spécification. Un tel objectif s'appuie sur l'existence de langages formels implantés qui permettent la description de l'architecture d'un système concurrent, la vérification de propriétés sur ce système et le raffinement vers un modèle plus concret (transformation). Ces ressources sont issues du monde du logiciel libre (Maude [Maude, 98],...) et de résultats de précédents travaux de l'équipe ArchLog (projet européen ArchLog, PI-ADL.Net) du laboratoire VALORIA. Les développements pourront se faire autour de la plate-forme de simulation robotique Robotics Studio [Microsoft Robotic Studio, 07]. Les missions seront donc ici exprimées dans un langage lié au domaine de la robotique (Domain Specific Language), MASL, qui permet à l'utilisateur final de spécifier ses missions. Une mission exprimée dans ce langage est ensuite transformée automatiquement en un exécutable sur une plate-forme cible suivant une composition hiérarchique et concurrente de services distants qui implantent des architectures robotiques réactives, délibératives ou hybrides.

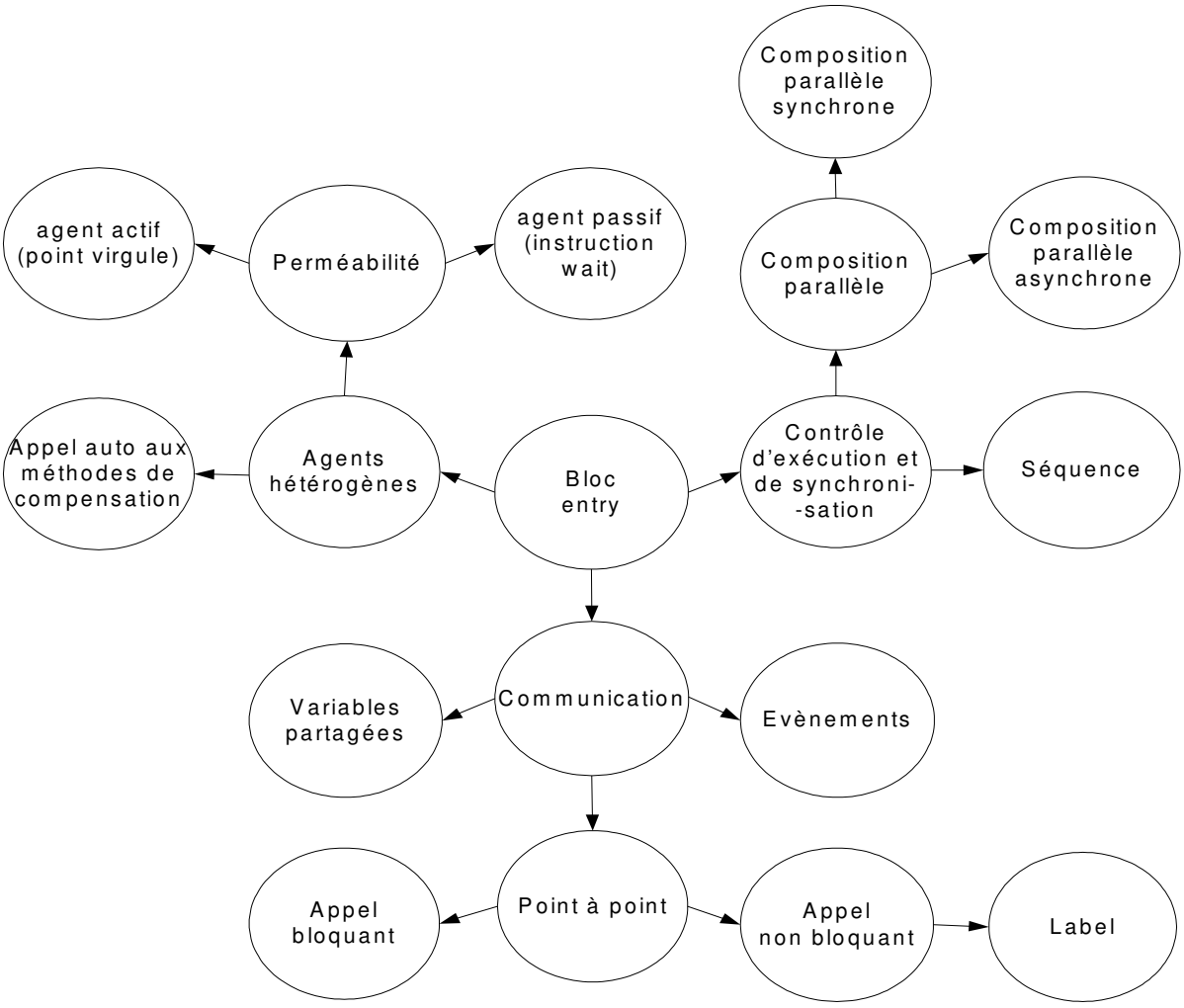

Figure 5 Carte de fonctionnalités de MASL : le bloc **`entry`** est au centre.

Nous souhaitons rendre la plateforme MASL conforme au standard FIPA [FIPA, 01] pour permettre l'interopérabilité 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. MASL étant une généralisation de HoRoCoL et se dernier visant à permettre au niveau du projet Robea MAAM l'utilisation complémentaire/substituable des approches Planification (*Path planning*) à l'aide des LCS (*Learning Classifier Systems*), résolution de modèles (*model resolution*), réseaux de neurones, algorithmes génétiques (*Emerging/learning*).

Du point de vue de l'interface de programmation, deux pistes d'améliorations sont envisagées. Nous souhaitons doter MASL d'une représentation graphique pour permettre au programmeur/utilisateur une programmation visuelle. Enfin, MASL permettant d'exprimer des missions de manière succincte, l'addition d'un module de traitement du langage naturel dans l'architecture rendra l'interaction avec l'utilisateur beaucoup plus simple. Ces pistes lui donnent la possibilité de modifier en ligne une mission.

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.

[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

[Brooks, 91] R. A. Brooks. *Intelligence without representation*. Artificial Intelligence, 47:139–159, 1991.

[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.

[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.

[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

[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

[Ferber & all, 00] Jacques Ferber et Olivier Gutknecht, *Operational semantics of Multi-agent Organizations*, LCNS Intelligent Agents VI, LNAI 1757, pp 205-217, 2000.

[Firby, 89] James Firby, *Adaptive Execution in Complex Dynamic Worlds*, PhD thesis, Yale University, 1989.

[Gat, 97] E. Gat. *ESL: a language for supporting robust plan execution in embedded autonomous agents.* In Proceedings of the IEEE Aerospace Conference, 1997.

[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)

[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, 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.

[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.

[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

[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

[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.

[Lötzsch, 04] M. Lötzsch. *XABSL - A Behavior Engineering System for Autonomous Agents*. Diploma thesis. Humboldt-Universität zu Berlin, 2004.

[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.

[Milner, 99] R. Milner, *Communicating and mobile systems : the π-calculus*, Cambridge university press, 1999.

[Mondada & all, 03] F. Mondada & all "*Swarm–bot : for concept to implementation*", IEEE/RSJ int conf on intelligent robots and systems IROS 2003

[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.

[Oquendo, 04] F. Oquendo, *π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures*, ACM Software Engineering Notes, Vol. 29, No. 3, pp. 15-28, May 2004

[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

[Ricordel, 01] Ricordel, Pierre-Michel, *Programmation Orientée Multi-Agents : Développement et Déploiement de Systèmes Multi Agents Voyelles*, thèse de doctorat, 2001.

[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

[Russel & all, 06] Stuart Russell, Peter Norvig, *Intelligence artificielle*, 2ème edition, Pearson Education France, Paris, 2006

[Simmons & all, 98] R. Simmons and D. Apfelbaum, "*A Task Description Language for Robot Control*," Proceedings Conference on Intelligent Robotics and Systems, October, 1998.

[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, 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.

Webographie

[FIPA, 01] FIPA '*Communicative Act Library Specification'*, Rapport technique XC00037H, 15 Août 2001, http://www.fipa.org/specs/fipa00037/.

[Java3D, 00] https://java3d.dev.java.net/

[JavaCC, 97] https://javacc.dev.java.net/

[JNI, 97] Java Native Interface : http://java.sun.com/javase/6/docs/technotes/guides/jni/

[Maude, 98] http://maude.cs.uiuc.edu/

[Microsoft Robotic Studio, 07] http://msdn.microsoft.com/fr-fr/robotics/default(en-us).aspx

[ODEJava, 04] OdeJava, *Open Dynamics Engine binding for Java*, http://odejava.org/OdejavaIntro/

[PMD, 04] http://pmd.sourceforge.net/

[SWIG, 95] Simplified Wrapper and Interface Generator : http://www.swig.org