Un module de gestion du monde pour un compagnon artificiel

Germain Lemasson

Rapport de stage

Un module de gestion du monde pour un compagnon artificiel

Germain Lemasson

Master Recherche Informatique

Systèmes intelligents auto-organisés

Encadrant : Dominique Duhaut

Février – Juin 2012

Résumé

Ce document présente le travail que j'ai effectué lors de mon stage au sein du laboratoire Lab-STICC. Il présente un module de gestion de monde qui complète l'architecture développée au sein du Lab-STICC. Cette architecture permet de programmer un ensemble de dispositif électronique.

Mots clés : **Intelligence Ambiante**, **Robotique**, **Ordonnancement**.

Abstract :

This document presents the work I have done in the Lab-STICC laboratory. It presents an arbitration manager module which completes the software achitecture develloped by the Lab-STICC. This architecture allows people programming an environment composed of many digital devices.

Key word: **Ambiant Intelligence**, **Robotic**, **Scheduling**.

REMERCIEMENTS

Je tiens à remercier mon encadrant, **Dominique Duhaut**, qui m'a proposé ce stage, qui m'a conseillé tous au long de sa réalisation et qui m'a offert des conditions de travail agréables.

Je tiens également à remercier **Céline Jost** qui m'a intégré au sein de ce projet, qui a toujours pris le temps de répondre à mes requêtes et de travailler avec moi.

Je remercie également **Thibaut Le Naour** et **Djamel Benferhat** pour avoir été des collègue sympa et m'avoir beaucoup renseigner sur la poursuite en thèse.

Je souhaite aussi remercier aussi **Vanessa André** pour avoir apporté de la bonne humeur dans le bureau.

Plan

PROBLEMATIQUE

De nos jours nous sommes entourés par de nombreux dispositifs électroniques. Ils ont des fonctionnalités de plus en plus complexes. Certains restent simples, comme un thermostat, un détecteur de fumée ou des volets automatiques. D'autres le sont moins comme certains dispositifs électroménagers qui ont de nombreuses options ou même les télévisions. De plus beaucoup commencent à avoir de multiples fonctionnalités comme les ordinateurs ou les smartphones. Bientôt nous aurons peut-être à notre disposition des dispositifs bien plus complexes tels que des robots. Le contrôle de tous ces dispositifs peut devenir vite compliqué. La plupart d'entre eux ont maintenant la possibilité de communiquer, branchés à un réseau interne ou même à internet. Grâce à cela il est alors possible de gérer au mieux cet ensemble de manière simple et cohérente. Des chercheurs imaginent comment cette gestion pourra être faite dans l'avenir. Ce domaine de recherche est appelé Intelligence Ambiante (AmI). Beaucoup de projets dans ce domaine ont pour but de créer des maisons intelligentes ou visent les centre médicaux afin d'aider le personnel dans leur tâche. D'autres projets en AmI s'intéressent au musée, centre commercial et en général tous les lieux où les personnes interagissent avec des dispositifs électroniques.

Le projet Robadom s'intéresse aux centres pour personnes âgées atteintes de trouble cognitif léger. Dans le cadre de ce projet une architecture a été développée. Elle a pour but d'aider le personnel et les patients dans leur vie quotidienne. Dans un tel environnement il y a plusieurs dispositifs mais il y a aussi plusieurs personnes en interaction avec ceux-ci. Les dispositifs électroniques de l'environnement peuvent être un ensemble de capteurs tel que des caméras, des détecteurs de mouvement ou des détecteurs de fumée. L'environnement peut contenir des télévisions, des ordinateurs, des robots, …. Un ensemble de logiciels a été conçu au Lab-STICC afin de pouvoir créer graphiquement des scénarios utilisant les différents dispositifs. Un scénario est un programme défini par un prestataire qui organise les réponses des différents dispositifs de l'environnement en fonction d'un ensemble de capteur. Il peut être écrit par le personnel médical, des amis, la famille du patients ou le patient lui-même. Ces personnes ont des intérêts différents, elles créent donc des scénarios différents. Ces scénarios sont ensuite exécutés en parallèle.

La question que je dois résoudre dans le cadre de mon stage est « Comment gérer les conflits et les priorités de l'ensemble des scénarios afin de répondre au mieux à la demande en fonction des disponibilités des actionneurs de l'environnement ? »

Dans une première partie nous verrons des travaux qui ont des liens avec la problématique. Ensuite nous découvrirons l'architecture développée au sein du Lab-STICC et quels sont les objectif du stage. Le chapitre 3 présente la solution que j'ai proposé afin de répondre aux différents objectifs. Le chapitre 4 montre une mise en application de cette solution.

1. TRAVAUX CONNEXES A LA PROBLEMATIQUE

1.1 INTELLIGENCE AMBIANTE

Le terme *intelligence ambiante* (AmI) est apparu pour la première fois lors d'un concours lancé par l'ISTAG (European Community's Information Society Technology Advisory Groups). Ducatel et coll. 2001 [1] propose plusieurs scénarios montrant comment l'intelligence ambiante pourrait être vécue en 2010, 9 ans après la publication. L'un des scénarios présente Maria une femme arrivant dans un pays où l'AmI est très développée. Elle possède un appareil l'identifiant et agissant sur l'environnement autour d'elle : enregistrement automatique une fois atterri à l'aéroport, ouverture de la porte de la voiture quand elle est assez près, guidage gps jusqu'à l'hôtel. Arrivé à l'hôtel l'appareil configure la chambre afin qu'elle corresponde au préférence de sa propriétaire. L'appareil est contrôlé par commande vocale uniquement. Nos smartphones tendent à cette approche mais nous n'en somme pas encore là.

De nombreux projets de recherche ont accepté le terme d'*intelligence ambiante* [2]. L'AmI consiste à créer des environnements subvenant au besoin des personnes en son sein.

Un des environnements privilégiés de l'AmI est la maison car c'est un lieu que l'on aimerait aussi confortable que possible. De nombreux projets se sont développés afin de créer des maisons « intelligentes » répondant aux besoins de ses habitants. Deux tendances de l'AmI sont la diminution des interactions directes entre les nombreux dispositifs et l'automatisation de certaines tâches. Le projet GENIO de Garate et coll. [3] développe une maison où l'ensemble des dispositifs électroménagers est connecté à un serveur central. Il est alors possible de commander ces dispositifs en discutant en langage naturel avec un majordome virtuel.

Figure 1. Maior-Domo representation

L'habitant doit toujours donner lui-même les ordres mais il interagit non plus avec un ensemble d'appareil mais une seule entité virtuelle qui sert d'interface. Deux autres projets MavHome [4] et iDorm [5] présentent des environnements contenant un ensemble de capteurs afin d'apprendre les habitudes de leur occupant et ainsi de pouvoir par la suite anticiper leur actions et les effectuer pour eux. Les actions en question sont basiques tel que l'ouverture, la fermeture de porte, l'allumage de la lumière, l'ouverture des volets. Le but est l'automatisation de toutes ces actions répétitives.

D'autres projets d'AmI visent des environnements destinés aux personnes âgées afin des les aider dans leur vie quotidienne. Robocare [6] est l'un d'eux. Le système connaît le planning de la personne âgée. Quelles sont les actions qu'elle doit effectuer, à quel moment et quelles sont leurs durées ? L'environnement est surveillé afin de vérifier que la personne respecte bien son planning (repas et prise de médicament). Le planning est recalculé en fonction des actions réelles de la personne et s'il devient insatisfaisant une alerte est lancée. Il existe d'autres projets dans l'aide à la personne âgée. Ils consistent principalement à surveiller les actions de la personne afin de vérifier qu'elle effectue les actions qu'elle doit faire et qu'elle ne se trouve pas dans des situations dangereuses (chutes, crise cardiaque).

Les techniques d'apprentissage permettent de réduire considérablement les tâches répétitives que doivent effectuer les personnes. Mais cela reste limité aux actions simples et courtes. L'architecture développée dans le projet Robadom permet d'automatiser des tâches plus compliquées à travers la création de scénarios par l'homme prenant en compte l'état de l'environnement. Les scénarios doivent pouvoir être créés par les personnes se trouvant dans l'environnement, donc novice en matière de programmation. Un scénario doit pouvoir rester simple. Cependant on souhaiterait pouvoir contrôler de nombreux dispositifs. C'est pour cela que l'architecture permet la création de plusieurs scénarios qui tournent en parallèle. Ces différents scénarios envoient des ordres aux dispositifs. On peut alors se trouver dans des cas de conflits. Il faut alors ordonner ces ordres. Les projets d'AmI se posent rarement ce problème d'ordonnancement. Il est alors intéressant de regarder ce qui se fait dans d'autres domaines.

1.2 ORDONNANCEMENT

Les deux domaines où le problème d'ordonnancement et de parallélisation est présent sont l'ordonnancement de processeur et la gestion de système de production.

L'ordonnancement de processeur consiste à assigner des tâches aux différents processeurs dans une architecture multi cœurs. Les algorithmes proposés cherchent à minimiser le temps d'exécution global. Ils estiment la durée que peut prendre une tâche sur un processeur et grâce à cela ils assignent et ordonnent l'ensemble des tâches. Les tâches sont des opérations de calcul. Elles peuvent être effectuées sur les différents processeurs cependant leur temps d'exécution peut varier d'un processeur à l'autre. Dans notre cas les tâches à effectuer ne peuvent être exécutées que par un sous ensemble des dispositifs. De plus nous le verrons plus tard nous ne pouvons pas estimer la durée des tâches à effectuer. Or de nombreux algorithmes se basent sur cette donnée. Un des algorithmes explorés est celui de Page et Naughton [7]. Ils utilisent un algorithme génétique permettant d'ordonnancer des tâches variées sur des processeurs différents. L'intérêt est d'un algorithme génétique est qu'il peut être modifiable afin de correspondre à nos contraintes (notamment en modifiant de la fonction de fitness). Cependant ce type d'algorithme reste lourd. Dans le cas de l'ordonnancement de processeur, c'est très efficace car il permet de gérer un grand nombre de tâches (10000) sur un grand nombre de processeur (50). Dans notre cas nous n'aurons pas plus d'une centaine de tâches sur un nombre de dispositifs réduit (une cinquantaine mais dont uniquement deux ou trois peuvent effectuer une tâche précise). Cette solution est certainement trop lourde pour l'utilisation que l'on veut en faire. L'ordonnancement de processeur a ses propres caractéristiques qui diffèrent des nôtres.

1.3 SYSTEME MULTIAGENT

Un système de production est plus proche de notre domaine. Il y a un ensemble de machine pouvant effectuer des tâches précises. Certaines machines existent en plusieurs exemplaire afin d'augmenter la productivité. C'est justement le but des systèmes mis en place : produire un maximum de pièce en un minimum de temps. L'ordonnancement de système de production doit être dynamique. Il doit prendre en compte les changement qui peuvent arrivé dans l'environnement (panne, nouvelle commande,…). Plusieurs chercher s'intéresse au système multi agent (SMA) pour réglé se type de problème. Par exemple Chang et coll. dans [8] présente un algorithme d'ordonnancement distribué basé sur les SMA. Leur but est d'alloué des tâches de maintenance à plusieurs ingénieurs ayant des compétences qui peuvent se recouper. Un protocole de négociation, basé sur le Contract Net Protocol [9], est utilisé afin de trouver une assignation optimale. Une fois encore une des données utilisées est la durée des tâches. Malgré cela certains points restent intéressants. Madureira et coll. [10] proposent un mécanisme de négociation pour les systèmes d'ordonnancement. Ce qui est intéressant ici n'est pas la négociation en elle-même car elle utilise les données de temps que nous ne disposons pas, mais plutôt l'architecture derrière ce mécanisme. C'est aussi une architecture multi agent. Le point que j'ai retenu est la création d'un agent pour chaque ressource utilisée (machine). Ces agents sont chargés d'ordonnancer les tâches qui leur sont attribuées sans se soucier des autres contraintes telles que la précédence qui sont gérées par d'autres agents.

L'ouverture à d'autres domaines permet uniquement de trouver des pistes de solutions. En effet certains objectifs sont les même mais les contraintes et les données disponibles dans le projet Robadom sont différentes. Le prochain chapitre détaillera ce point.

2 L'ARCHITECTURE MGI DU LAB-STICC

2.1 ARCHITECTURE EXISTANTE

Le projet Robadom s'inscrit dans un axe thématique lié au développement de technologies ou de services innovants au domicile des patients atteints de la maladie d'Alzheimer. Il prévoit la conception d'applications interactives pour un humain et un ensemble de dispositifs numériques (cas de la domotique par exemple).

L'objectif de l'architecture informatique développée est de permettre à plusieurs acteurs différents de programmer simplement cet ensemble de dispositifs. On obtient un certain nombre de programmes qui tournent en parallèle et qui agissent sur les dispositifs gérant l'interaction avec la personne.

Ci-dessous la présentation de MGI, le Modèle Générique pour l'Interaction. C'est un ensemble de logiciel permettant de gérer l'interaction entre un homme et un ensemble de machines qui l'entoure. Nous l'appellerons Architecture informatique pour environnement interactif

multimodal. Ces machines peuvent être des robots, mais également des avatars virtuels, qu'ils soient déployés sur un écran d'ordinateur, de télévision, de téléphone portable etc.

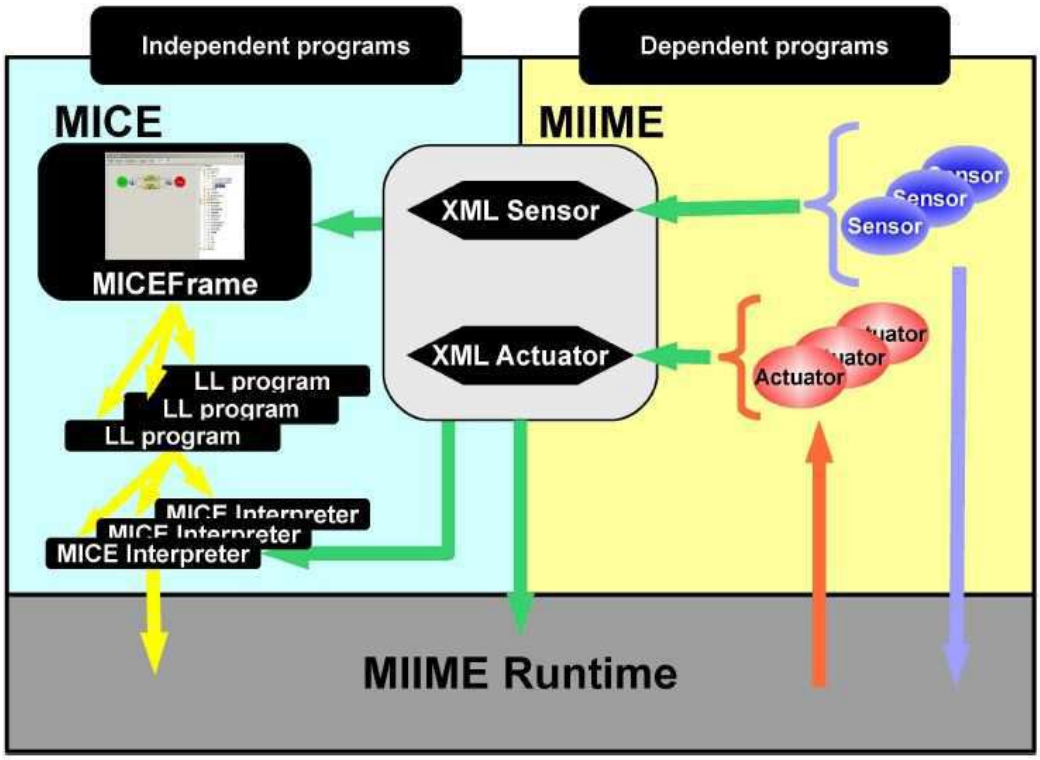


Figure 2. Modèle Générique pour l'Interaction

MGI est composé d'un système de pilotage de l'interaction, que l'on appelle MICE (Machines Interaction Control in their Environement) et d'une architecture informatique MIIME (Modules for Interaction and Individualization of Machines with Emotions).

MGI est organisé de la façon suivante :

MICEFrame qui permet de concevoir une application liant les informations issues des capteurs (**sensor**) aux actionneurs de l'environnement (**actuator**). MICEFrame se configure automatiquement pour permettre la programmation de l'environnement à partir des fichiers XML qui décrivent ces capteurs et ces actionneurs.

MICEFrame transforme cette programmation en un programme informatique écrit dans un langage spécifique : le langage LL.

Le programme généré est lu et interprété par **MICE Interpreter**. Lors de l'interprétation, **MICE Interpreter** utilise la couche d'exécution **MIIME Runtime**, à la fois pour se servir des informations qui viennent des capteurs mais aussi pour envoyer les informations aux actionneurs.

MIIME Runtime permet le transport des informations.

MICEFrame, **MICE Interpreter** et **MIIME Runtime** connaissent l'environnement grâce à la connaissance apporté par les fichiers XML qui décrivent les capteurs et les actionneurs.

2.1.1 MIIME : ENVIRONNEMENT D'EXECUTION

MIIME – Modules d'Interaction et d'Individualisation des Machines Emotionnelles est l'architecture informatique qui permet de mettre en oeuvre MICE. Elle permet de mettre en relation des capteurs, des programmes et des objets communicants.

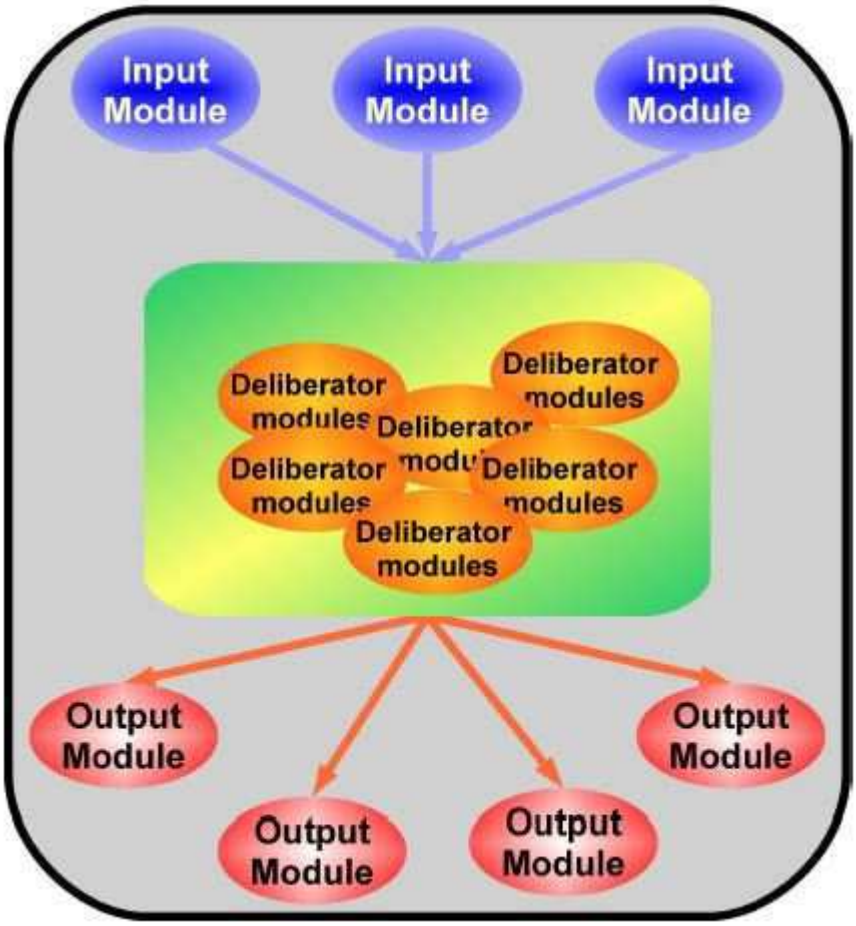


Figure 3. Organisation de MIIME

La Figure 3 illustre l'organisation de MIIME en montrant qu'elle est composée de trois types de modules qui communiquent entre eux. Un module est un programme informatique autonome, qui est spécialisé dans une tâche particulière.

Input Module (module d'entrées) : ils sont responsables d'un capteur et envoient des informations liées à l'analyse du capteur.

Deliberator Module (module de délibération) : ces modules ne sont responsables ni d'un capteur, ni d'une entité. Ce sont des programmes informatiques qui sont responsables de la gestion de l'interaction homme-machine, c'est le cœur du programme cognitif.

Output Module (module de sortie) : ils sont responsables d'un actionneur. Ils sont chargés de la communication avec cet actionneur et de transmettre les actions à effectuer.

Les modules reçoivent des informations d'autres modules et envoient des informations aux autres modules. La communication est standardisée pour faciliter l'ajout de nouveaux modules.

Ainsi, les messages transitant entre les modules sont sous la forme d'une chaîne XML au format MIIME et sont constitués de la façon suivante :

```
01.  <io type="">
02.    <subject></subject>
03.    <from></from>
04.    <to></to>
05.    <size></size>
06.    <time></time>
07.    <frequency></frequency>
08.    <message></message>
09.  </io>
```

Le message io possède deux types d'information :

Les informations, relatives au transport des données, qui sont la partie fixe du format MIIME : subject, from, to, size, time et frequency. Chaque module peut avoir accès à ces informations de façon certaine. Toutes les balises sont obligatoires sauf time et frequency.

Les informations qui apportent la sémantique du message sont contenues dans la balise message. Elles ne sont accessibles et connues que par chaque module individuellement. Leur format est spécifique à chaque module et non standardisé, contrairement aux informations susmentionnées.

Les messages io sont inspirés des emails, mais adaptés aux besoins de l'architecture :

- type : représente le type de données et permet aux modules qui reçoivent le message de savoir rapidement s'ils doivent les traiter ou non, et comment les traiter. Il peut y avoir :
    - input : ce sont des messages en provenance des capteurs
    - deliberator : ce sont des messages de calcul
    - output : ce sont des actions à transmettre aux entités
    - info : ce sont des messages spécifiques au transit des messages
- subject : apporte une précision sur le contenu du message et permet aux modules de savoir comment interpréter le sens du message contenu dans la balise message. On trouve par exemple, « event », « speech », « scenario » etc.
- from : indique l'expéditeur du message
- to : indique le destinataire du message
- time : indique l'heure à laquelle le message a été envoyé
- frequency : indique si ce message revient périodiquement ou non en donnant la valeur de la fréquence
- size : indique la taille du message
- message : contient le message que le module doit interpréter.

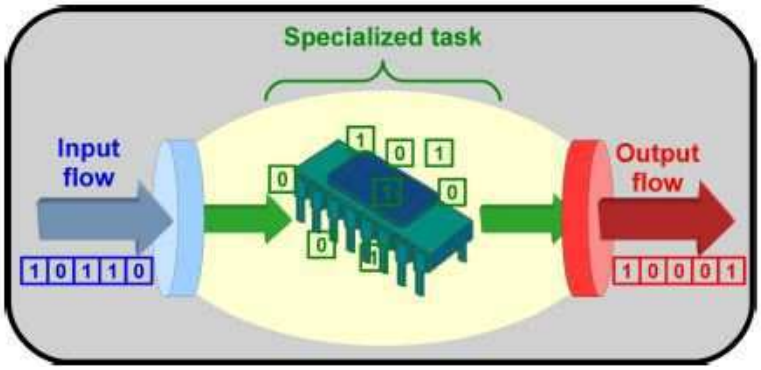


Figure 4. Les entrées et sorties d'un module

Comme le montre la Figure 4, les modules ont une entrée et une sortie. Les informations qui arrivent en entrée viennent d'autres modules, tandis que les informations envoyées sur la sortie arrivent dans l'entrée d'autres modules. On parle alors des liens entres les modules, à savoir qu'un lien entre un module A et un module B indique que la sortie de A est envoyée à l'entrée de B.

Les modules ne travaillent pas ensembles et ne se connaissent pas les uns les autres, ils communiquent seulement par les messages io à travers le système. Ainsi chaque module reçoit un message en entrée, exécute une tâche qui peut modifier ce message et envoie ce message sur sa sortie. Les Input Module n'ont pas d'entrée et les Output Module n'ont pas de sortie.

MICE est une implémentation de cette architecture permettant de contrôler les différents dispositifs de l'environnement.

2.1.2 MICE : CONCEPTION ET EXECUTION DE PROGRAMMES EN ENVIRONNEMENT INTERACTIF

MICE - Machines Interaction Control in their Environment (Pilotage de l'Interaction des Machines dans leur Environnement) est un outil permettant de contrôler l'interaction entre l'homme et un ensemble de dispositifs. Par exemple, dans le cas de la domotique, les dispositifs peuvent être : le thermostat du chauffage, un indicateur de luminosité, un détecteur de présence, une caméra, des volets roulants, des lumières...

Il faut noter deux types de dispositifs :

- les capteurs, qui permettent de donner des informations sur l'environnement, comme précédemment cités le thermostat, l'indicateur de luminosité...
- les actionneurs, qui effectuent des actions et qui réagissent aux données envoyées par les capteurs, comme les volets roulants et les lumières qui réagissent au capteur de luminosité par exemple.

MICE offre la possibilité de programmer le comportement des actionneurs en fonction des données envoyées par les capteurs, à travers une interface graphique MICEFrame.

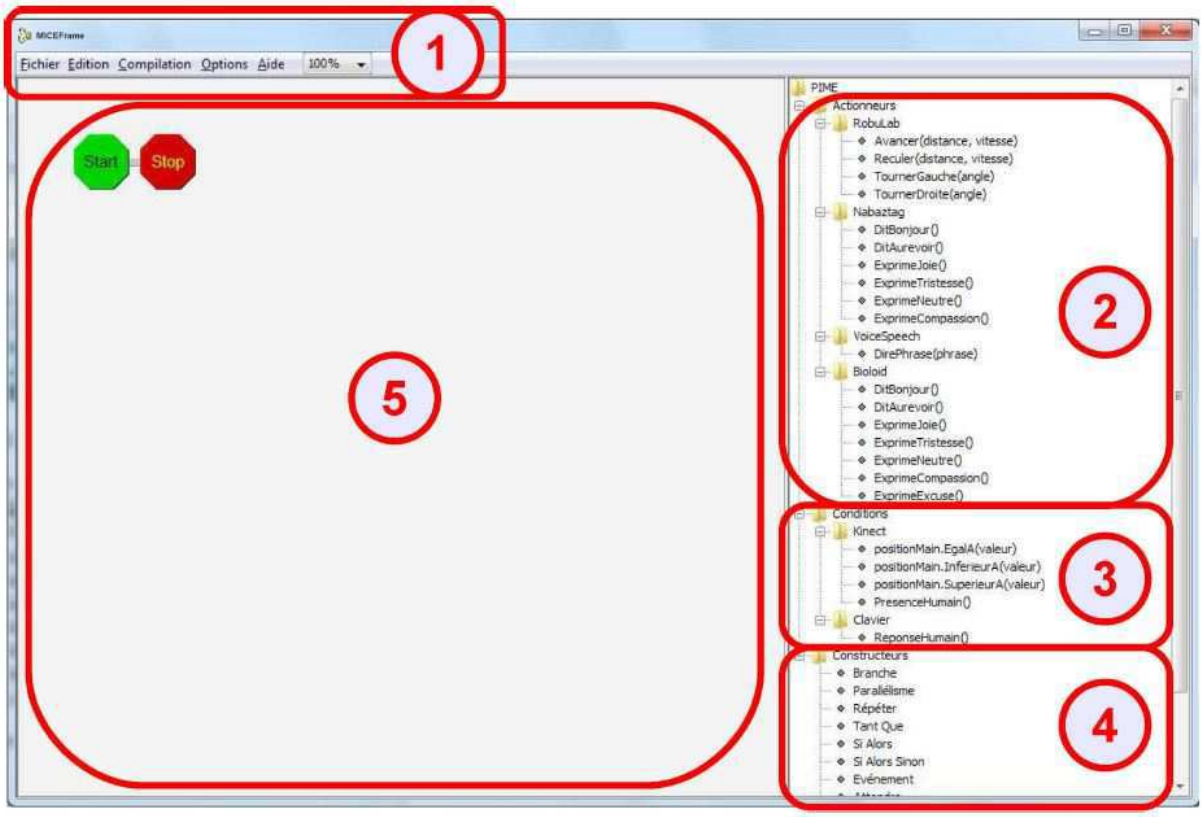


Figure 5. Interface de pilotage : MICEFrame

La Figure 5 présente l'organisation de l'interface de pilotage qui se décompose en cinq zones. La zone 1 est la barre de menu de l'application. Elle propose d'enregistrer ou de charger des programmes, d'exporter le programme sous forme d'images, d'exporter le programme sous forme d'un fichier qui est utilisé pour gérer l'interaction et autres fonctionnalités de base. La zone 2 montre la liste des actionneurs actuellement connectés et permet d'accéder à toutes les fonctionnalités qu'ils proposent. Par exemple : pour l'actionneur « volets roulants », les fonctionnalités peuvent être « ouvrir », « fermer » etc. La zone 3 montre la liste des capteurs actuellement connectés et permet d'accéder à la liste des données qu'ils peuvent envoyer. Par exemple : pour le capteur « luminosité », les données peuvent être « jour », « nuit », sombre » etc. La zone 4 propose les outils de programmation qui permettent de gérer l'interaction. Ces outils sont décrits dans le paragraphe suivant. La zone 5 est la fenêtre de visualisation du programme en cours de construction.

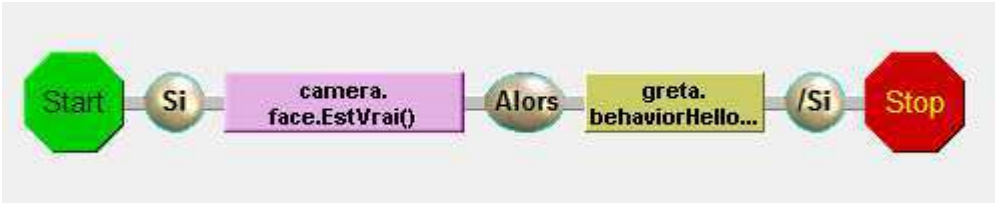


Figure 6. Exemple de scénario

La Figure 6 montre un scénario simple pouvant être créé avec MICEFrame. La traduction est la suivante. Si le capteur camera détecte un visage alors l'actionneur Greta doit exécuter l'action « behaviorHello ».

Un scénario est présenté sous la forme d'un graphe. Les nœuds de se graphe sont des outils de programmation (les cercles), des actions (le rectangle à droite) ou des conditions (le rectangle à gauche)

Les outils de programmations sont les suivants :

- Si alors
- Si alors sinon
- Tant que
- Répéter X fois
- Attendre
- Evènement
- Parallélisme
- Branche
- Break

Avec l'outil Evènement, on attend tant que la condition n'est pas remplie.

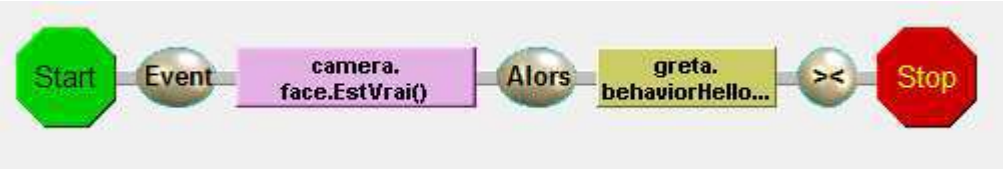


Figure 7. Scénario utilisant l'outil Evènement

Le scénario de la Figure 7 se traduit par : Tant que l'on ne détecte pas de visage on attend. Au moment où un visage est détecté, l'actionneur Greta doit exécuter l'action « behaviorHello ».

L'outil Parallélisme permet de demander que plusieurs actions s'effectuent en parallèle. L'outil Branche rajoute une branche à l'outil Parallélisme.

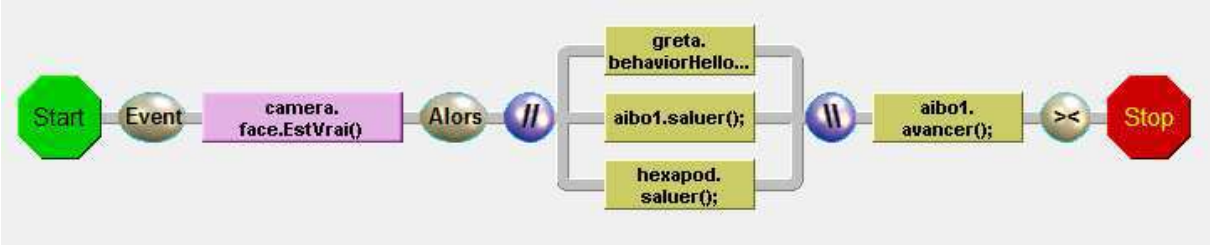


Figure 8. Scénario utilisant l'outil Parallélisme

Le scénario de la Figure 8 se traduit par : Tant que l'on ne détecte pas de visage on attend. Au moment où un visage est détecté, l'actionneur Greta doit exécuter l'action « behaviorHello », l'actionneur aibo1 doit exécuter l'action « saluer » et l'actionneur hexapod doit exécuter l'action « saluer ». Une fois que ces trois actions sont terminées, l'actionneur aibo1 doit exécuter l'action « avancer ».

Les autres outils de programmation correspondent sensiblement à ceux des langages procéduraux connus.

Une fois qu'un scénario est créé, il faut l'interpréter.

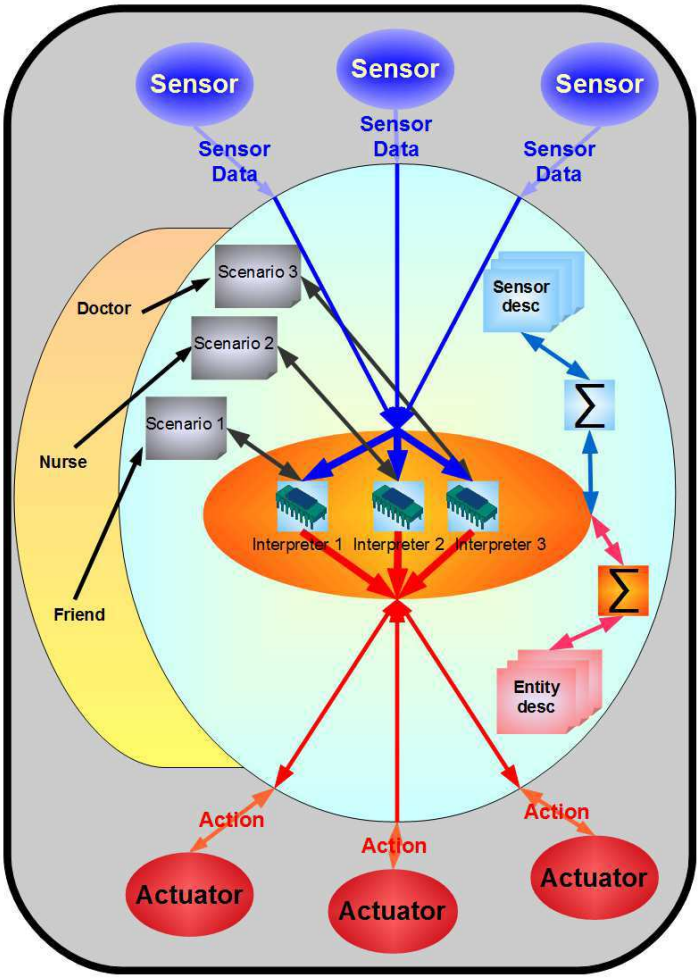


Figure 9. Implémentation de MIIME

La Figure 9 montre l'utilisation de l'architecture permettant d'interpréter les scénarios.

Chaque scénario est interprété par un MICE Interpreter. Il reçoit en entrée les données de module Input Sensor et renvoi des ordres à des modules Output Actuator.

Les MICE Interpreter sont des modules indépendants tournant en parallèles. On peut en arrêter ou en lancer de nouveaux sans que cela perturbe ceux qui tournent.

Les Sensor en bleu sont des Input Module de MIIME qui sont lié à un capteur de l'environnement (camera, détecteur, etc.) Les Sensor sont décrit par des fichiers de descriptions accessibles par les MICE Interpreter.

Ci-dessous un exemple de fichier de description

```
01.   <sensor id="camera">
02.       <perception name="qrcode" min="0" max="50" type="number" />
03.       <perception name="face" min="0" max="1" type="boolean" />
04.   </sensor>
```

Le fichier contient l'identifiant du Sensor et les valeurs qu'il peut envoyer ainsi que leur type.

Un MICE Interpreter utilise les données reçues pour dérouler le scénario qu'il interprète. Lorsque qu'il arrive sur une action, il envoie un message au module Actuator correspondant.

Les Actuator en rouge, sur la Figure 9, sont des Output Module de MIIME et sont lié à un actionneur de l'environnement (robot, avatar, volets roulants, etc.) Ils ont aussi des fichiers de descriptions accessibles par les MICE Interpreter.

```
01.  <entity id="jauge1">
02.      <fonction name="fill">
03.        <argument id="value" min="0" max="100" type="number" />
04.        <ordre>
05.          <fill>value</fill>
06.        </ordre>
07.      </fonction>
08.      <fonction name="vacate">
09.        <ordre>
10.          <vacate/>
11.        </ordre>
12.      </fonction>
13.  </entity>
```

Un fichier de description contient l'Id de l'actionneur ainsi que la liste des actions qu'il est capable d'effectuer. Ici jauge1 est capable de se remplir d'une certaine valeur passée en argument ou de se vider.

Les actionneurs sont des dispositifs physiques, les actions qu'ils sont capables d'exécuter ne sont pas instantanées. Par exemple un volet roulant met du temps à s'ouvrir complétement.

Dans l'architecture le choix a été fait que les Actuator attaché aux dispositifs envoient un message indiquant que l'action est terminée. Ce message prend le chemin inverse du message demandant l'action. Ainsi l'interpréteur ne continue son exécution que lorsqu'il reçoit ce message de fin d'exécution. Dans le cas d'un parallélisme les autres branches continuent leur exécution.

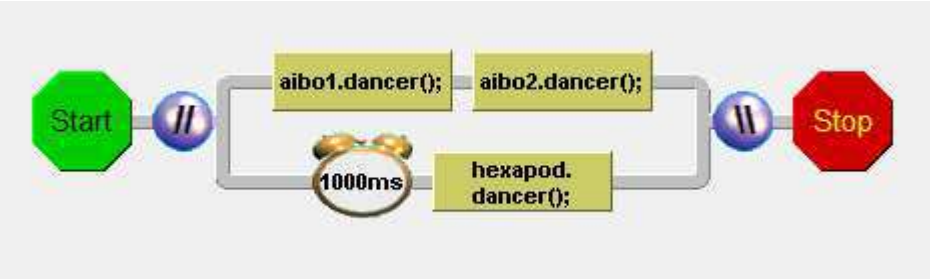


Figure 10. Enchaînement d'actions

Dans la Figure 10 l'interpréteur enverra l'ordre dance à l'actionneur aibo2 que lorsqu'il aura reçu le message de fin d'exécution de l'actionneur aibo1. Cependant quel que soit la durée de l'ordre « danser » de l'actionneur aibo1, il enverra un ordre « danser » à l'actionneur hexapod au bout de une seconde.

2.2 LE MODULE DE GESTION DU MONDE DEMANDE

L'architecture présente est fonctionnelle. Cependant dans certains cas des problèmes ont été constatés.

Les interpréteurs tournent en parallèle et chacun d'eux envoient des ordres aux différents actionneurs. Ils sont indépendant les uns des autres, ils ne savent pas ce que font les autres. Si deux des scénarios interprétés utilisent un même actionneur, il est possible qu'à un moment donné cet actionneur reçoive un ordre alors qu'il est déjà en train d'en exécuter un autre. Il pourrait simplement le mettre en attente mais une gestion prenant en compte l'environnement est plus appropriée.

Parmi les différents actionneurs, plusieurs d'entre eux offrent parfois des fonctionnalités similaires. Au moment de la création d'un scénario, le prestataire ne sait pas forcement lequel choisir. Il souhaite que l'ordre soit effectué par le meilleur actionneur. Cette notion de meilleur dépend aussi de l'état de l'environnement au moment où l'ordre est demandé.

Le dernier problème rencontré est lié à l'outil de programmation Parallélisation. Il permet de paralléliser des exécutions et notamment de paralléliser les actions de plusieurs actionneurs. En plus de cette parallélisation, le prestataire souhaite peut-être que ces actionneurs soient synchronisés et qu'ils ne soient pas perturbés par d'autre scénario.

L'objectif du stage est de créer un module de gestion qui arbitre les différents ordres envoyés par les interpréteurs afin de répondre aux problèmes énoncés.

La Figure 11 montre où se trouve le gestionnaire dans l'architecture. Il se trouve entre les interpréteurs et les actionneurs. Il doit résoudre trois types de problème décrit dans les paragraphes ci-dessous.

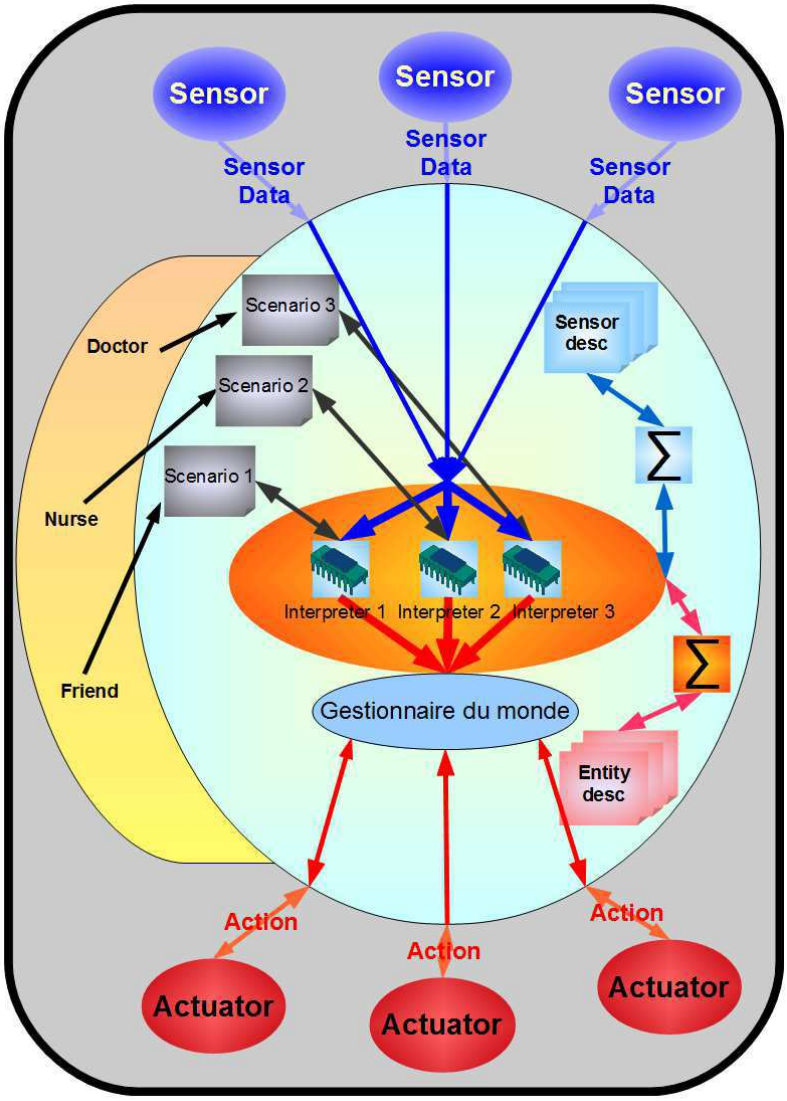


Figure 11. Ajout du gestionnaire

2.2.1 GESTION DE CONFLIT DANS LE CAS MULTI SCENARIOS

Le gestionnaire doit empêcher l'arrivée d'ordre sur un actionneur alors qu'il est occupé. Il doit mettre des ordres en attente. Lorsque l'actionneur se libère il doit choisir judicieusement lequel des ordres en attente il doit envoyer. Les ordres viennent de différents scénarios. Les scénarios sont écrits par différentes personnes. Elles peuvent être le patient, un ami, la famille, une infirmière ou un médecin. Les scénarios qu'ils écrivent sont plus ou moins importants selon leurs statuts. Le scénario écrit par le médecin peut par exemple être plus important que celui écrit par un ami du patient. De plus les scénarios correspondent à des situations différentes : jeu, prise de médicament, cas d'urgence (chute). Ces deux critères permettent de déduire une priorité entre les scénarios. Le gestionnaire doit arbitrer les ordres en utilisant cette priorité.

2.2.2 ACTIONNEUR GENERIQUE

Le second problème que doit permettre le gestionnaire est d'ajouter une fonctionnalité à MICE. Lors de la création des scénarios, l'utilisateur choisit spécifiquement quel actionneur doit

effectuer l'ordre demandé. Parmi les actionneurs disponibles, il arrive parfois que plusieurs d'entre eux puissent exécuter le même ordre. Dans ce cas à la création du scénario, l'utilisateur ne sait pas forcement lequel choisir. La création d'un actionneur générique capable d'effectuer l'ensemble des actions exécutables par les actionneurs réels. Les ordres qui lui sont destinés, sont récupérés par le gestionnaire. C'est lui qui choisi quel est l'actionneur réel le plus apte à remplir cette action. Le premier critère est de choisir un actionneur libre. Le second critère utilisé est un indicateur de qualité calculé dynamiquement selon l'environnement. Par exemple deux robots ont la capacité de se déplacer dans une pièce. Le robot, le plus proche de la pièce, peut être considéré comme de meilleure qualité. Le calcul de cet indicateur n'est pas l'objet de cette étude. Il est considéré accessible à n'importe quel moment par le gestionnaire.

2.2.3 SYNCHRONISATION DE PLUSIEURS ACTIONNEURS.

Dans le cas où l'outil Parallélisation parallélise uniquement des actions, le gestionnaire doit permettre de synchroniser leurs exécutions. Il doit s'assurer que les actionneurs impliqués soient tous libres en même temps et il doit empêcher leur accès tant que toutes les actions ne sont pas terminées.

2.3 CONTRAINTES AUXQUELLES EST SOUMIS LE GESTIONNAIRE DU MONDE

Le gestionnaire doit accomplir les trois tâches expliquées ci-dessus en respectant deux contraintes liées à l'architecture utilisée :

- La première d'entre elles est l'arrivée en temps réel des actions. Nous l'avons vu dans le chapitre expliquant MICE. Les interpréteurs envoient les ordres au moment où ils doivent être exécutés. Le gestionnaire n'a donc aucune vision globale des scénarios mais reçoit uniquement les ordres qui doivent être exécutés.
- La seconde contrainte est liée aux modules actionneurs qui sont de type hétéroclite. Le gestionnaire a accès à leurs fiches de descriptions mais elles renseignent uniquement le format des messages reçus et les fonctions qu'ils proposent. Cependant ceux-ci peuvent être de toutes sortes, des simples volets roulants au robot multifonctions. Les actions qu'ils peuvent effectuer sont parfois complexes et leur temps d'exécution peut varier selon l'environnement. Par exemple si on demande à un robot d'aller dans une pièce, le temps qu'il va mettre pour y aller dépendra de son lieu actuel et des obstacles qu'il rencontrera. On ne saura l'achèvement de son opération que lorsque qu'il aura envoyé le message de fin d'exécution. Ceci est la contrainte la plus forte pour la création du gestionnaire. Nous l'avons vu précédemment l'ordonnancement de tâches est connu dans d'autres domaines tel que l'ordonnancement de processeurs et les systèmes de productions. Cependant on connaît toujours le temps d'exécution des tâches ou il est possible de l'estimer. Le gestionnaire doit se passer de cette donnée.

3 SOLUTION PROPOSEE

Pour prendre en compte les trois points décrits en 2.2, j'ai construit un algorithme de gestion du monde.

Il a été développé en trois étapes. Chacune de ces étapes correspond à un des problèmes rencontrés :

- 1ère étape : Gestion de conflit dans le cas multi scénarios
- 2ème étape : Actionneur générique
- 3ème étape : Synchronisation de plusieurs actionneurs

L'étape 2 améliore l'algorithme de l'étape 1 afin de gérer un actionneur générique. Et l'étape 3 améliore l'algorithme afin de pouvoir synchroniser des actions.

Pour plus de clarté, dans ce rapport je présente l'algorithme de chaque étape et pas seulement l'algorithme final.

3.1 GESTION DE CONFLIT DANS LE CAS MULTI SCENARIOS

La première étape est de gérer l'exécution parallèle des scénarios et les conflits qui sont susceptibles d'arriver. Deux scénarios sont considérés en état de conflit si un actionneur reçoit l'ordre d'un scénario alors qu'il exécute déjà l'ordre d'un autre scénario. La solution proposée est de temporiser l'arrivée des ordres au niveau des actionneurs. Rappelons que la durée des actions n'est pas disponible. L'algorithme se base uniquement sur les deux évènements suivants :

- Arrivée d'un nouvel ordre venant d'un des scénarios en cours.
- Arrivée d'un message indiquant qu'un ordre a fini de s'exécuter.

L'algorithme reprend l'idée de [8]. Pour chaque module Actuator connecté (cf. Figure 12), un objet Actuator Manager est créé pour chaque module Actuator connecté. Il sera chargé de s'occuper uniquement des messages destinés à son actionneur ou en provenance de celui-ci.

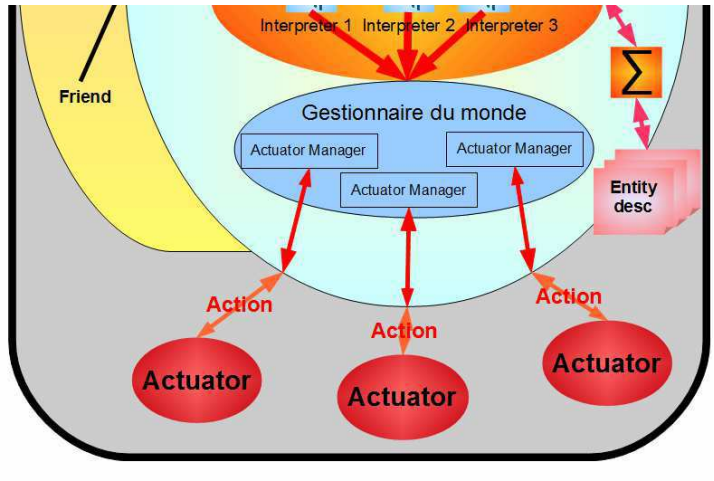


Figure 12. Actuator Manager

La Figure 13 présente le diagramme de classe du gestionnaire à l'étape 1.

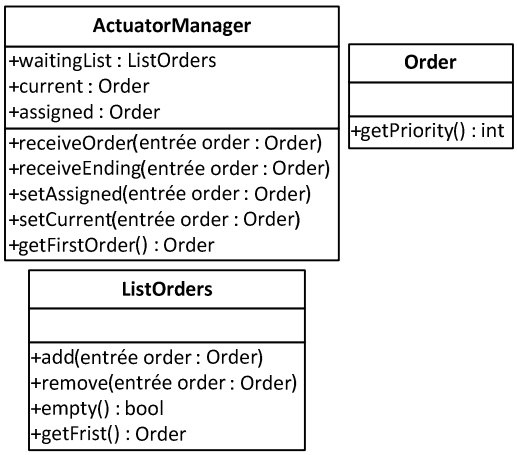


Figure 13. Diagramme de classe étape 1

Un Actuator Manager possède 3 attributs :

- waitingList : une liste non ordonnée des ordres en attente
- current : l'ordre qui est en cours d'exécution sur l'actionneur
- assigned : le prochain ordre qui devra être exécuté.

Et possède 4 fonctions

- receiveOrder(Order order) : fonction appelée à l'arrivée d'un nouvelle ordre
- receiveEnding(Order order) : fonction appelée à l'arrivée d'un message de fin d'éxecution
- setAssigned(Order order) : fonction affectant le champ assigned

- setCurrent(Order order) : fonction affectant le champ current et envoyant l'ordre à l'actionneur
- getFirst() : Order : cette fonction renvoie l'ordre le plus prioritaire contenu dans waitingList

Un objet Order contient la fonction suivante :

- getPriority() : elle donne la priorité de cette ordre en fonction du scénario d'où il provient.

Les fonctions de la liste waitingList sont les suivantes.

- add(Order order) : ajoute l'ordre à la fin de la liste
- remove(Order order) : supprime l'ordre de la liste s'il y est présent
- empty() : retourne vrai si la liste est vide, faux sinon
- getFirst() : retourne le premier élément de la liste. Le premier à avoir été ajouté.

3.1.1 ARRIVEE D'UN NOUVEL ORDRE VENANT D'UN DES SCENARIOS EN COURS

A l'arrivée d'un nouvel ordre au Gestionnaire du monde par un des interpréteurs (Figure 12), il est redirigé directement à l'Actuator Manager correspondant à sa cible. Ci-dessous la fonction recevant ce nouvel ordre dans un Actuator Manager :

```
01.  receiveOrder(Order order)
02.  {
03.   if(assigned==null)// S'il n'y a pas d'ordre en attente
04.   {
05.          if(current==null) //Si l'actionneur est libre
06.          {
07.                setCurrent(order); //L'ordre est envoyer à l'actionneur
08.          }else //Sinon
09.          {
10.                setAssigned(order); //L'ordre est le plus prioritaire en attente
11.          }
12.   }else
13.   {
05.          //Si le nouvel ordre est plus prioritaire que le prochain devant être
     lancer, il prend sa place
14.          if(order.getPriority()> assigned.getPriority())
15.          {
16.                waitingList.add(assigned);//l'ordre est mis en liste d'attente,
     il n'est plus le plus prioritaire
17.                setAssigned(order);
18.          }
19.          else //Sinon il est mis en attente
20.          {
21.                waitingList.add(order);
22.          }
23.   }
24.  }
```

3.1.2 ARRIVEE D'UN MESSAGE INDIQUANT QU'UN ORDRE A FINI DE S'EXECUTER

La seconde partie de l'algorithme est la réception d'un message de fin d'exécution.

```
01.  receiveEnding(Order order)
02.  {
06.   // Si des ordres sont en attente le plus prioritaire est dans le champ
     assigned, si assigned est null alors il n'y a aucun ordre en attente.
03.
04.  if(assigned!=null) //si un ordre est en attente
05.   {
06.          setCurrent(assigned); //On envoie directement l'ordre
07.          //On cherche ensuite le nouvel ordre le plus prioritaire
07.          Order assignedOrder = getFirstOrder();
08.          if(assignedOrder != null)
09.          {
10.                setAssigned(assignedOrder);
11.          }
12.   }
13.  }
14.
15.  Order getFirstOrder()
16.  {
17.   //si la liste n'est pas vide, elle est parcourue.
18.   if(!waitingList.empty())
19.   {
20.          //bestOrder est un tampon contenant le meilleur ordre de la liste
21.          Order bestOrder = waitingList.getFirst();
22.          foreach(Order order in waitingList)
23.          {
24.                if(order.getPriority()> assigned.getPriority())
25.                {
26.                      bestOrder=order;
27.                }
28.          }
29.          return bestOrder;
30.   }
31.   else
32.   {
33.          return null;
34.   }
35.  }
```

L'emploi d'une liste non ordonnée permet l'utilisation de priorité dynamique pour les scénarios. En effet l'algorithme parcourt une seule fois la liste et uniquement lorsque cela est nécessaire. En utilisant une liste ordonnée celle-ci aurait dû être réordonnée à chaque changement de l'environnement ce qui n'est absolument pas optimal. L'environnement peut contenir de très nombreux capteure et chacun d'eux envoient plusieurs valeurs susceptibles de changer la priorité des scénarios.

Dans cette première étape l'algorithme permet la gestion de plusieurs scénarios en temporisant l'arrivée des ordres. Les ordres en conflit sont triés selon la priorité des scénarios les ayants émis.

3.2 ACTIONNEUR GENERIQUE

La seconde étape complète le gestionnaire afin de permettre l'utilisation d'un actionneur générique. Les ordres destinés à cet actionneur virtuel doivent être assignés à l'actionneur le plus apte à remplir la tâche.

J'ai rajouté un Actuator Manager un peu différent des autres. Il gère tous les ordres destinés à l'actionneur générique. Je l'ai appelé GenericActuatorManager (GAM) (cf. Figure 14).

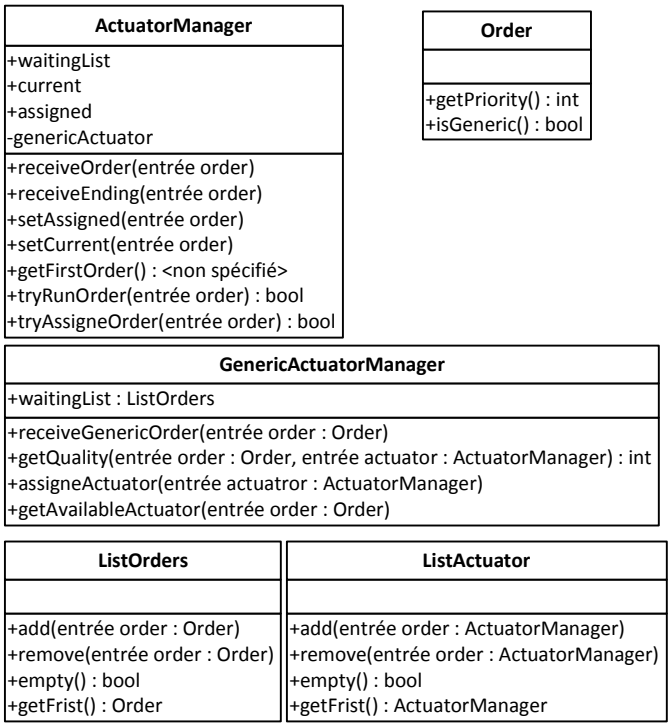


Figure 14. Diagramme de classe étape 2

Ci-dessous le détaille des fonction du GAM :

- waitingList : une liste des ordres générique en attente
- Integer getQuality(ActuatorManager actuator, Order order) : cette fonction retourne la qualité d'exécution de l'ordre order sur l'actionneur de actuator au moment où la fonction est appelée. Cette fonction n'est pas détaillée car elle n'est pas l'objet de l'étude.
- List getAvailableActuator(Order order) : cette fonction retourne la liste des actuator ayant la possibilité de remplir l'ordre donné en paramètre. Cela est possible grâce au fichier de description des actionneurs. Le gestionnaire connaît la liste des ordres que chacun peut effectuer. A partir de ces fichiers il crée une table de hachage contenant des couples (ordre, liste d'actionneur). La table renvoie la liste des ActuatorManager capables d'exécuter l'ordre passé en clé. Ainsi getAvailableActuator(Order order) a une complexité de O(N) dans le pire des cas, où N est le nombre d'ordres différents que permet l'ensemble des actionneurs.

3.2.1 ARRIVEE D'UN NOUVEL ORDRE VENANT D'UN DES SCENARIOS EN COURS

Les ActuatorManager sont modifiés pour permettre au GAM de leur attribuer des ordres génériques.

La première modification est l'ajout de la fonction tryRunOrder. Elle permet de tenter d'exécuter l'ordre en paramètre et renvoie le résultat de cette tentative.

```
01.  boolean tryRunOrder(Order order)
02.  {
03.   //Il ne doit pas y avoir d'ordre en attente si on veut envoyer l'ordre.
04.   if(assigned==null)
05.   {
06.          if(current==null)
07.          {
08.                setCurrent(order);
09.                return true;
10.          }else
11.          {
12.                return false
13.          }
14.   }
15.   else
16.   {
17.          return false
18.   }
19.  }
```

La seconde modification est l'ajout de la fonction tryAssigneOrder (Order order) qui de la même façon tente d'assigner l'ordre passé en paramètre et renvoi le résultat de cette tentative.

```
01.  boolean tryAssigneOrder(Order order)
02.  {
03.   if(assigned==null)
04.   {
05.          setAssigned(order);
06.          return true;
07.   }else
08.   {
09.          //Si le nouvel ordre générique est plus prioritaire que celui
     actuellement assigné
10.          if(order.getPriority()> assigned.getPriority())
11.          {
12.                //Si l'ordre actuellement assigné est aussi un ordre générique
     il doit être mis dans la liste d'attente des ordres génériques.
13.                if(assigned.isGeneric())
14.                {
15.                      genericActuator.waitingList.add(assigned);
16.                }
17.                else
18.                {
19.                      waitingList.add(assigned);
20.                }
21.                setAssigned(order);
22.                return true;
23.          }
24.          else
25.          {
26.                return false;
27.          }
28.   }
29.  }
```

Ces deux nouvelles fonctions permettent au GAM de passer des ordres aux ActuatorManager.

Ci-dessous la fonction appelée lors de la réception d'un ordre générique (sans cible) :

```
01.  receiveOrder(Order order)
02.  {
03.   // availableActuators contient les actionneurs pouvant éxecuter order
04.   ListActuator availableActuators = getAvailableActuator(order);
05.
06.   //Si au moins un actionneur peut exécuter l'ordre
07.   if(!availableActuators.empty())
08.   {
09.          //sort est une fonction de trie standard dont le comparateur utilise
     la fonction getQuality(ActuatorManager actuator, Order order)
```

```
10.           sort(availableActuators, order);
11.           //le premier élément de availableActuators est l'actuator de meilleure
      qualité.
12.           boolean running=false;
13.           //On tente d'exécuter l'ordre dans l'ordre des qualités.
14.           foreach(ActuatorManager actuator in availableActuators)
15.           {
16.                 if(actuator.tryRunOrder(order))
17.                 {
18.                       running=true;
19.                       break;
20.                 }
21.           }
22.
23.           if(!running)
24.           {
25.                 //Si l'éxécution a échoué, on tente d'assigné l'ordre
26.                 boolean assigned=false;
27.                 foreach(ActuatorManager actuator in availableActuators)
28.                 {
29.                       if(actuator.tryAssigneOrder(order))
30.                       {
31.                             assigned=true;
32.                             break;
33.                       }
34.                 }
35.                 if(!assigned)
36.                 {
37.                       //Si l'assignation a échoué il est mis en liste
      d'attente.
38.                       waitingList.add(order)
39.                 }
40.           }
41.    }
42.   }
```

Il faut aussi adapter receiveOrder(Order order) de 3.1.1

```
01.   receiveOrder(Order order)
02.   {
03.    if(assigned==null)
04.    {
05.           if(current==null)
06.           {
07.                 setCurrent(order);
08.           }else
09.           {
10.                 setAssigned(order);
11.           }
12.    }else
13.    {
14.           if(order.getPriority()> assigned.getPriority())
15.           {
16.                 //Si l'ordre actuellement assigné est aussi un ordre générique
      il doit être mis dans la liste d'attente des ordres génériques.
17.                 if(assigned.isGeneric())
18.                 {
19.                       genericActuator.waitingList.add(assigned);
20.                 }
21.                 else
22.                 {
23.                       waitingList.add(assigned);
24.                 }
25.                 setAssigned(order);
26.           }
27.           else
28.           {
29.                 waitingList.add(order);
30.           }
31.    }
32.   }
```

3.2.2 ARRIVEE D'UN MESSAGE INDIQUANT QU'UN ORDRE A FINI DE S'EXECUTER

Lors de la réception d'un message de terminaison d'ordre, l'ordre réassigné doit être le plus prioritaire de la liste d'attente de l'ActuatorManager **ET** de la liste d'attente du GAM.

C'est donc GAM qui s'occupe maintenant de la réassignation avec la fonction assigneActuator(ActuatorManager actuator).

On adapte en conséquence la fonction receiveEnding(Order order) de 3.1.2

```
01.  receiveEnding(Order order)
02.  {
03.   if(assigned!=null)
04.   {
05.          setCurrent(assigned);
06.          genericActuator.assigneActuator(this);
07.   }
08.  }
```

Ci-dessous la fonction qui réassigne un actionneur quand il devient libre.

```
01.  assigneActuator(ActuatorManager actuator)
02.  {
03.   //On récupère le meilleur ordre de la liste d'attente de l'actionneur.
04.   Order bestorder=actuator.getFirstOrder();

05.   //Si la liste était vide
06.   if(bestorder==null)
07.   {
08.          //Si la seconde liste est vide il n'y a pas de réassignation.
09.          if(waitingList.isEmpty())
10.          {
11.                return;
12.          }
13.          else
14.          {
15.                //Sinon initialise bestorder avec le premier ordre de la liste
16.               bestorder= waitingList.getFirst();
17.          }
18.   }
19.
20.   //La liste des ordres générique est parcouru afin de récupérer l'ordre le
     plus prioritaire et de meilleure qualité. La priorité des scénarios est
     prépondérante à la qualité des ordres.
21.   foreach(Order order in waitingList)
22.   {
23.          if(actuator.canRun(order))
24.          {
25.                if(order.getPriority()> bestorder.getPriority())
26.                {
27.                      bestorder=order;
28.                }
29.                else if (order.getPriority()== bestorder.getPriority())
30.                {
31.                      if(getQuality(order, actuator)> getQuality(bestorder,
     actuator))
32.                      {
33.                            bestorder=order;
34.                      }
35.                }
36.          }
37.   }
38.   //Le meilleure ordre est assigné à l'actuator
39.   actuator.setAssigned(bestorder);
```

```
40.    if(bestorder.isGeneric())
41.    {
42.           waitingList.remove(bestorder);
43.    }
44.    else
45.    {
46.           actuator.waitingList.remove(bestorder);
47.    }
48.
49.  }
```

Cette fonction a une complexité dans le pire des cas en O(2N+M) ou O(N+2M), où N est la taille de la liste d'attente de l'ActuatorManager et M la taille de la liste d'attente du GenericActuatorManager, selon dans quelle liste est fait la suppression finale.

Dans cette seconde étape, l'algorithme est capable de gérer un actionneur générique tous en évitant les conflits entre scénario.

3.3 SYNCHRONISATION DE PLUSIEURS ACTIONNEURS.

L'outil Parallélisation des scénarios permet de lancer plusieurs traitements en parallèle. Lorsque ces traitements parallèles sont des exécutions d'ordre, le parallélisme est considéré comme synchronisé. Les exécutions doivent commencer en même temps sur les différents actionneurs impliqués. De plus les actionneurs ne doivent pas être disponibles tant que le parallélisme n'est pas terminé : l'exécution de tous les ordres est terminée.

La Figure 15 est un parallélisme synchronisé, chaque branche est constituée d'une action. La Figure 16 n'en est pas un car la seconde branche n'est pas une simple action.

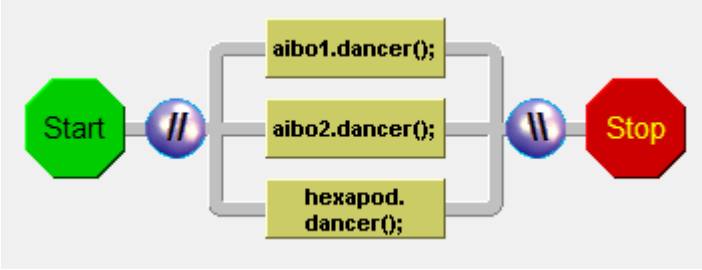


Figure 15. Parallélisme synchronisé

Figure 16. Parallélisme non synchronisé

Rappelons que le gestionnaire n'a pas connaissance de la structure des scénarios. Il ne peut donc pas savoir si les actions qu'il reçoit font partie d'un parallélisme synchronisé ou non, seul le MICE Interpreter interprétant le scénario le sait. Celui-ci a donc été modifié afin qu'il envoie un message au début de ce parallélisme. Le message contient un identifiant et le nombre de branches (d'ordres) contenue dans le parallélisme. Les ordres impliqués contiennent l'identifiant du parallélisme.

Il est supposé qu'un actionneur ne se trouve pas sur plusieurs branches. Un actionneur ne peut être synchronisé avec lui-même. L'actionneur générique n'apparaît sur aucune des branches.

Ma solution est la suivante.

A l'arrivée d'un message de parallélisme synchronisé (PS), l'algorithme crée un objet GroupOrder(GO). Les ordres apparentant à ce groupe, lui sont transmis au GO mais pas à l'actionneur cible. Une fois tous les ordres du groupe arrivés, le GO les envoie au ActuatorManager à qui ils sont destinés. Les ActuatorManager sont modifiés afin de ne pas envoyer d'ordre appartenant à un groupe mais uniquement tenter de les assigner. C'est le GroupOrder qui fait l'envoi lorsque tous les ordres sont assignés et tous les actionneurs sont libres. Il s'occupe aussi de libérer les actionneurs quand tous les ordres sont terminés. Le GAM est aussi modifié afin de pouvoir réassigner un ensemble d'actionneurs en même temps.

La Figure 17 présente le nouveau diagramme de classe. Je détaille ensuite chaque nouvelle propriété.

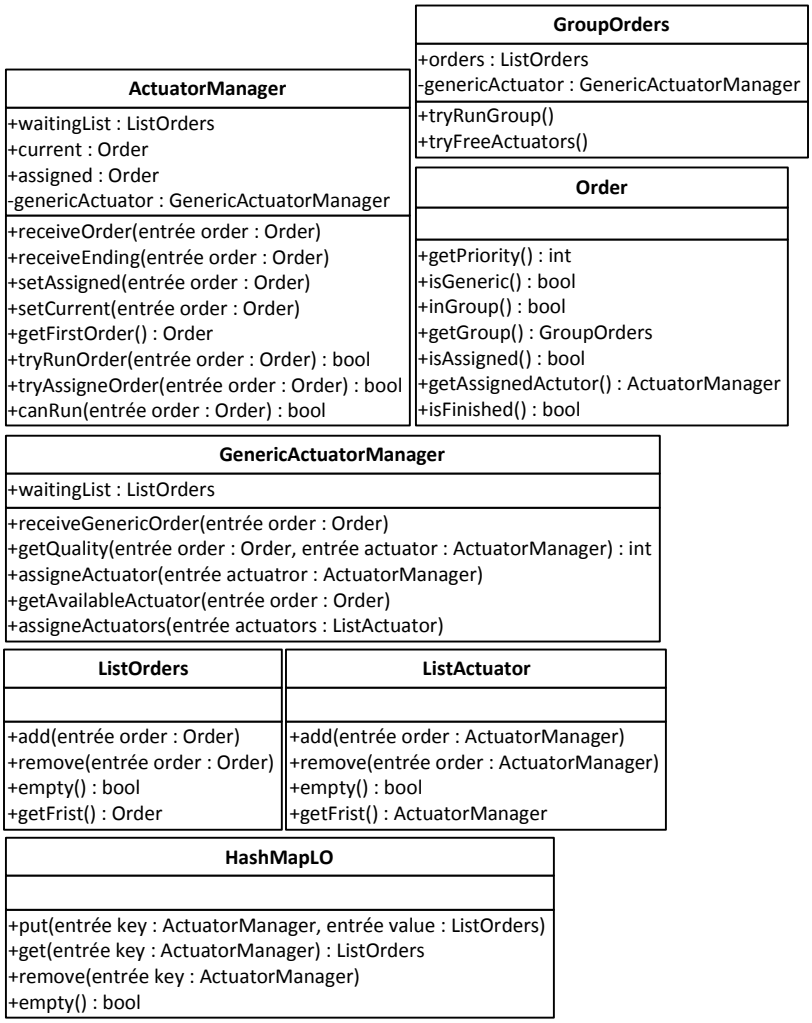


Figure 17. Diagramme de classe étape 3

Les objets Order sont modifiés afin de prendre en compte leur appartenance à un groupe. Ils ont de nouvelles fonctions :

- inGroup() : renvoie vrai si cet ordre est dans un PS.
- getGroup() : renvoie le GroupOrder dans lequel se trouve l'ordre
- isFinished() : renvoie vrai si l'exécution de l'ordre est terminée
- isAssigned() : renvoie vrai si l'ordre est assigné à son Actuator Manager (champ assigned)
- getAssignedActuator() : retourne l'Actuator Manager où l'ordre est assigné

Les ActuatorManager ont aussi une nouvelle fonction :

- canRun(Order order) : renvoie vrai si l'ordre passé en paramètre est un des ordres disponible sur l'actionneur.

Un GroupOrder a les propriétés suivantes :

- orders : la liste des ordres du groupe

- tryRunGroup() : tente d'envoyer l'ensemble des ordres si les conditions le permettent. Cette fonction est appelée à chaque fois que l'on assigne un des ordres du groupe.
- tryFreeActuators() : tente de libérer les actionneurs si tous les ordres sont terminés. Cette fonction est appelée à chaque fois qu'un ordre se termine.

3.3.1 ARRIVEE D'UN NOUVEL ORDRE VENANT D'UN DES SCENARIOS EN COURS

Ci-dessous l'algorithme de la fonction tryRunGroup() de la classe GroupOrders

```
01.  tryRunGroup()
02.  {
03.   boolean runnable = true;
04.   foreach(Order order in orders) // Les ordres du groupe sont parcourus
05.   {
06.          if(!order.isAssigned())// Si un ordre n'est pas assigné, le groupe ne
     peut être exécuté
07.          {
08.                 runnable=false;
09.                 break;
10.          }
11.          else
12.          {
13.                 //Si l'ordre est assigné mais que l'actionneur est occupé, le
     groupe ne peut être exécuté
14.                if(order.getAssigneActuator().current!=null)
15.                {
16.                       runnable=false;
17.                       break;
18.                }
19.          }
20.   }
21.   if(runnable) //Si le groupe peut être exécuté.
22.   {
23.          // actuators contient l'ensemble des actionneurs impliqués dans la
     synchronisation
24.          List actuators;
25.          foreach(Order order in orders) //Tous les ordres sont envoyés
26.          {
27.                 actuators.add(order.getAssignedActuator());
28.                 order.getAssignedActuator().setCurrent(order);
29.                 order.getAssignedActuator().setAssigned(null);
30.          }
31.          //Les actionneurs impliqués demandent une réassignation.
32.          genericActuator.assigneActuator(actuators);
33.   }
34.  }
```

receiveOrder(Order order) de Actuator Manager a été modifié pour ne pas envoyer les ordres groupés.

```
01.  receiveOrder(Order order)
02.  {
03.   if(assigned==null)
04.   {
05.          if(current==null && !order.inGroup())
06.          {
07.                 setCurrent(order);
08.          }else
09.          {
10.                 setAssigned(order);
11.          }
12.   }else
13.   {
14.          if(order.getPriority()> assigned.getPriority())
15.          {
16.                 if(assigned.isGeneric())
17.                 {
18.                        genericActuator.waitingList.add(assigned);
```

```
19.                 }
20.                 else
21.                 {
22.                       waitingList.add(assigned);
23.                 }
24.                 setAssigned(order);
25.                 //Si l'ordre assigné appartient à un groupe on tente d'exécuté
    le groupe.
26.                 if(order.inGroup())
27.                 {
28.                       order.getGroup().tryRunGroup();
29.                 }
30.           }
31.           else
32.           {
33.                 waitingList.add(order);
34.           }
35.    }
36.  }
```

L'autre modification importante est l'ajout au GAM d'une fonction réassignant plusieurs actionneurs en même temps. En effet à l'exécution d'un groupe ou à sa libération, il faut réassigner un ensemble d'actionneur. L'algorithme pourrait appeler plusieurs fois la fonction assigneActuator(ActuatorManager actuator) de l'étape 2. Or l'ordre dans lequel sont réassigné les actionneurs, a une importance. Un ordre générique en attente pourrait être assigné à un actionneur (premier réassigné) alors qu'il aurait une meilleure qualité sur un actionneur réassigné plus tard. La réassignation doit se faire globalement. Ci-dessous le code la fonction assigneActuators(ListActuator actuators) qui prend en paramètre une liste d' ActuatorManager.

```
01.  assigneActuator(ListActuator actuators)
02.  {
03.    // bestOrderForActuators est une table de hachage qui donne pour chaque
    actionneur une liste des actions qu'il peut effectuer.
04.    HashMapLO bestOrderForActuators;
05.    foreach(ActuatorManager actuator in actuators)
06.    {
07.          ListOrders orders;
08.          //La liste des ordres est initialise avec l'ordre le plus prioritaire
    de la liste d'attente l'ActuatorManager.
09.          if(actuator.getFirstOrder()!=null)
10.          {
11.                orders.add(actuator.getFirstOrder());
12.          }
13.          bestOrderForActuators.add(actuator,orders);
14.    }
15.    //Cette boucle remplis les listes d'ordre par actionneur. Un ordre générique
    peut se trouver dans plusieurs listes.
16.    foreach(Order order in waitingList)
17.    {
18.          foreach(ActuatorManager actuator in actuators)
19.          {
20.                if(actuator.canRun(order))
21.                {
22.                      bestOrderForActuators.get(actuator).add(order);
23.                }
24.          }
25.    }
26.    //Cette boucle trie les liste d'ordre de la même façon que sont triés les
    actionneurs dans la fonction receiveOrder(Order order) du GMA
27.    foreach(ActuatorManager actuator in actuators)
28.    {
29.          sort(bestOrderForActuators.getList(actuator), actuator);
30.    }
31.    //Cette boucle assigne un à un les actionneurs.
32.    while(actuators.isEmpty())
33.    {
```

```
34.
35.          Order bestActuator= actuators.getFirst();
36.          if(bestOrderForActuators.getList(bestActuator).isEmpty())
37.          {
38.                Order bestOrder=
     bestOrderForActuators.getList(bestActuator).getFirst();
39.                //La boucle compare les premiers éléments de chaque liste afin
     d'obtenir l'ordre le plus prioritaire et de meilleur qualité sur l'ensemble
     des ordres en attente
40.                foreach(ActuatorManager actuator in actuators)
41.                {
42.                      if(!bestOrderForActuators.getList(actuator).empty())
43.                      {
44.                            Order order =
     bestOrderForActuators.getList(actuator).getFirst();
45.                            if(order.getPriority()> bestOrder.getPriority())
46.                            {
47.                                  bestOrder=order;
48.                                  bestActuator=actuator;
49.                            }
50.                            else if (order.getPriority()==
     bestOrder.getPriority())
51.                            {
52.                                  if(getQuality(order, actuator)>
     getQuality(bestOrder, bestActuator))
53.                                  {
54.                                        bestOrder=order;
55.                                        bestActuator=actuator;
56.                                  }
57.                            }
58.                      }
59.                }
60.                // bestOrder est assigné à l'actionneur sur lequel il est
     meilleur
61.                bestActuator.setAssigned(bestOrder);
62.                //S'il appartient à un groupe on vérifie si ce groupe peut être
     lancé
63.                if(bestOrder.inGroup())
64.                {
65.                      bestOrder.getGroup().tryRunGroup();
66.                }
67.                //l'ordre est retiré de sa liste d'attente
68.                if(bestOrder.isGeneric())
69.                {
70.                      waitingList.remove(bestOrder);
71.                }
72.                else
73.                {
74.                      bestActuator.waitingList.remove(bestOrder);
75.                }
76.                //Il est aussi retiré des listes des autres actionneurs
77.                foreach(ActuatorManager actuator in actuators)
78.                {
79.
      bestOrderForActuators.getList(actuator).remove(bestOrder);
80.                }
81.
82.          }
83.          //L'actionneur est retiré des actionneurs à assigner.
84.          actuators.remove(bestActuator);
85.
86.    }
87.
88.  }
```

Cette fonction a une complexité élevée. Rappelons que les environnements visés sont un centre de soins pour personnes âgées ou une maison. Il est rare de vouloir synchroniser plus d'une dizaine d'actionneurs. De plus il y a peu de chance qu'ils aient tous de nombreux ordres en attentes.

3.3.2 ARRIVEE D'UN MESSAGE INDIQUANT QU'UN ORDRE A FINI DE S'EXECUTER

A l'arrivée d'un message de fin d'exécution, si l'ordre appartient à un groupe, c'est le groupe qui détermine si l'Actuator doit être libéré.

Ci-dessous l'adaptation de la fonction receiveEnding(Order order) de Actuator Manager.

```
01.  receiveEnding(Order order)
02.  {
03.   if(order.inGroup())
04.   {
05.         //Si l'ordre appartient à un groupe un vérifie que tout le groupe est
     terminé.
06.         order.getGroup().checkFinished();
07.   }
08.   else
09.   {
10.         if(assigned!=null)
11.         {
12.               setCurrent(assigned);
13.               genericActuator.assigneActuator(this);
14.         }
15.   }
16.  }
```

Ci-dessous la fonction vérifiant que le groupe est terminé.

```
01.  tryFreeActuators()
02.  {
03.   boolean finished = true;
04.   //La boucle vérifie que chaque ordre est terminé.
05.   foreach(Order order in orders)
06.   {
07.         if(!order.isFinished())
08.         {
09.               finished=false;
10.               break;
11.         }
12.   }
13.   if(finished) //Si c'est le cas
14.   {
15.         List actuators;
16.         //La boucle libère, lance les ordres qui étaient en attente sur les
     actionneurs
17.         foreach(Order order in orders)
18.         {
19.               actuators.add(order.getAssignedActuator());
20.               Order assigned = order.getAssignedActuator().assigned;
21.               if(assigned!=null)
22.               {
23.                     if(assigned.inGroup())
24.                     {
25.                           assigned.getGroup().tryRunGroup()
26.                     }
27.                     else
28.                     {
29.                           order.getAssignedActuator().setCurrent(assigned);
30.                     }
31.               }
32.         }
33.         //Les actionneurs libérés sont réassignés.
34.         genericActuator.assigneActuator(actuators);
35.   }
36.  }
```

Avec cette étape finale le gestionnaire gère les trois objectifs souhaités.

4 MISE EN ŒUVRE DU GESTIONNAIRE DU MONDE

Afin de tester l'algorithme de gestion du monde que j'ai développé, une expérimentation a été effectuée au concours Robofesta (http://crdp2.ac-rennes.fr/robofesta/) se déroulant à Lorient le 23 mai. Cette expérimentation a mis en œuvre 4 scénarios dans un environnement contenant trois capteurs et trois actionneurs.

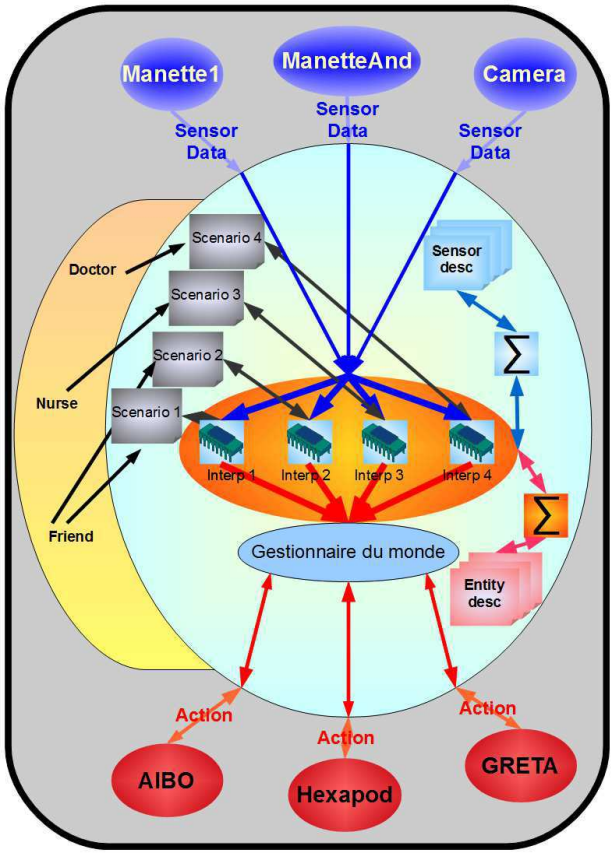


Figure 18. Schéma de l'expérimentation

4.1 MATERIEL UTILISE

Pour effectuer cette expérimentation, j'ai implémenté trois modules Sensor et trois modules Actuator. Les transferts de message dans l'architecture se font par socket tcp. Ce type de connexion m'a permis de créer des modules sur différentes plateformes (Windows, Android) et dans des langages différents (Java, C++). Le serveur MIIME (cf. chapitre 2) est un programme java.

4.1.1 CAMERA

Le premier module que j'ai implémenté est le Sensor camera. Il est capable de détecter le visage d'une personne et de lire des QR code (code barre à deux dimensions). La Figure 19 montre la détection de visage.

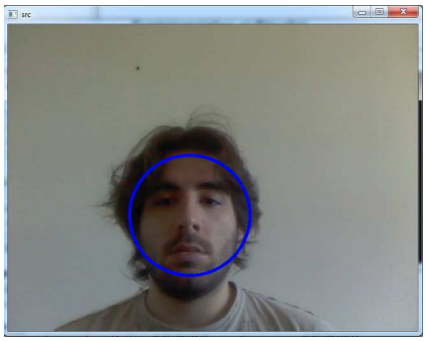


Figure 19. Détection Visage

Son fichier de description est le suivant.

```
01.   <sensor id="camera">
02.       <perception name="qrcode" min="0" max="50" type="number" />
03.       <perception name="face" min="0" max="1" type="boolean" />
04.   </sensor>
```

Je l'ai implémenté en C++. Afin de détecter les visages en temps réel j'ai utilisé la bibliothèque graphique OpenCV et plus précisément sa classe CascadeClassifier permettant la détection d'objets. Pour la lecture de code bar j'ai utilisé la bibliothèque libdecodeqr de Joseph Holsten. J'ai combiné ces deux bibliothèques dans un seul et même module.

4.1.2 MANETTE DE JEU

Le second module que j'ai développé simule une manette de jeu (Figure 20).

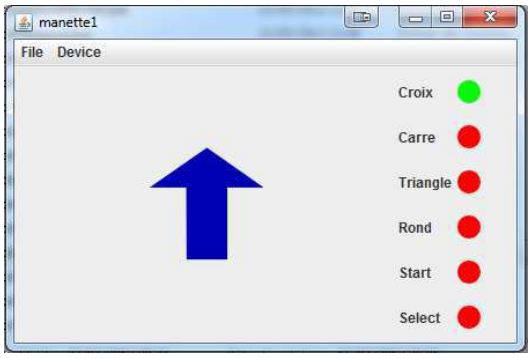


Figure 20. Module manette de jeu

Son fichier de description est le suivant.

```
01.   <sensor id="manette1">
02.       <perception name="direction" min="0" max="4" type="number" />
03.       <perception name="croix" min="0" max="1" type="boolean" />
04.       <perception name="rond" min="0" max="1" type="boolean" />
05.       <perception name="triangle" min="0" max="1" type="boolean" />
06.       <perception name="carre" min="0" max="1" type="boolean" />
07.       <perception name="start" min="0" max="1" type="boolean" />
08.       <perception name="select" min="0" max="1" type="boolean" />
09.     </sensor>
```

Il envoie une direction et l'état de 6 boutons.

Je l'ai développé en Java. Il récupère les évènements clavier. Cependant grâce à la librairie FFJoystick(http://boat.lachsfeld.at/ffjoystick4java/), le module peut être contrôlé par une vrai manette de jeux si on le souhaite.

Figure 21. Manette de jeu utilisé

4.1.3 MANETTE ANDROID

J'ai développé un second module de manette qui fournit les mêmes informations que le précédent. Toutefois ce module est une application smartphone android (Figure 22). Je l'ai développé pour la version 2.2.3 et 4.0 d'android. Il suffit de connecter en réseau le smartphone et le pc où se trouve le serveur MIIME et initialiser la connexion. La direction de la manette se contrôle avec l'accéléromètre du smartphone.

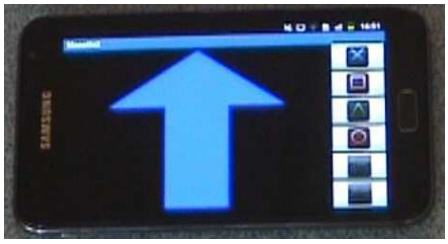


Figure 22. Manette Android

Son fichier de description est le suivant.

```
01.  <sensor id=" manetteAnd">
02.      <perception name="direction" min="0" max="4" type="number" />
03.      <perception name="croix" min="0" max="1" type="boolean" />
04.      <perception name="rond" min="0" max="1" type="boolean" />
05.      <perception name="triangle" min="0" max="1" type="boolean" />
06.      <perception name="carre" min="0" max="1" type="boolean" />
07.      <perception name="start" min="0" max="1" type="boolean" />
08.      <perception name="select" min="0" max="1" type="boolean" />
09.    </sensor>
```

4.1.4 AIBO SONY

J'ai ensuite développé un module permettant de piloter un robot chien AIBO de Sony (Figure 23).

Il est capable de 6 actions.

- Avancer()
- Reculer()
- TournerADroite()
- TournerAGauche()

- Stop()
- Saluer()

Figure 23. Chien AIBO Sony

Les chiens AIBO de Sony peuvent se contrôler à distance en wifi grâce à la plateforme logicielle Urbi. J'ai tout d'abord crée les mouvements qui m'intéressaient dans le robot avec le langage de programmation Urbiscript. Pour cela je me suis inspiré des programmes de déplacement donnés en exemple par la plateforme Urbi. J'ai ensuite créé un programme en C++ permettant de se connecter au robot et de lui envoyer les ordres voulus.

4.1.5 HEXAPODE

Un second robot est utilisé comme actionneur. Il a la même fonction que l'AIBO mais c'est un hexapode.

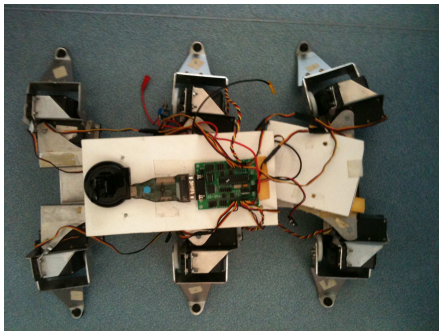


Figure 24. Hexapode

Le robot est connecté au PC avec de l'USB sans fils. Le module communique avec la carte électronique par liaison série. (J'utilise un convertisseur USB-Serie).

4.1.6 GRETA

Le dernier module actionneur communique avec un avatar virtuel animé GRETA (Figure 25). Le module existait déjà à mon arrivé au laboratoire. Je lui ai cependant rajouté les deux fonctions qui m'intéressaient.

- behaviorHello() : fait un salut de la main en disant le texte suivant : « Hello come play with us »
- ExplainSynchro() : fait un cercle avec ces bras en disant le texte suivant : « We are all synchronized »

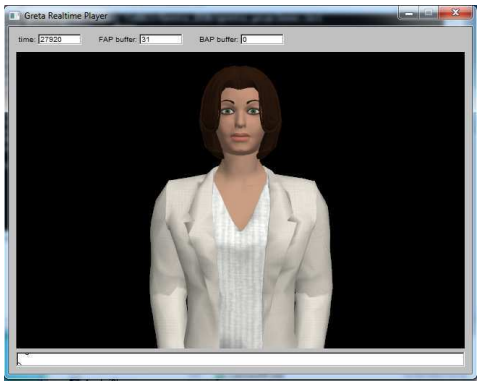


Figure 25. Avatar Greta

4.2 SCENARIOS

Les quatre scénarios que j'ai écrits sont les suivants :

- Scénario 1 : Pilotage d'un robot générique. Ce scénario utilise la manette pour envoyer des ordres de déplacement à l'actionneur générique.
- Scénario 2 : Greta dit bonjour quand la camera détecte un visage.
- Scénario 3 : Pilotage d'un robot spécifique. L'utilisateur utilise le smartphone pour choisir un robot, AIBO ou hexapode, et le pilote.
- Scénario 4 : Greta et les robots exécutent chacun une action synchronisée avec les autres, au moment où la camera lit un code barre.

Cf. Annexe pour voir les voir dans MICE Frame.

Les priorités des scénarios et la qualité des actionneurs sont des indicateurs dont le calcul ne fait pas partie des objectifs. Ils sont tous de même indispensable pour tester l'algorithme.

Les priorités des scénarios sont fixes. Elles sont les suivantes :

- scénario 1 : 10
- scénario 2 : 10
- scénario 3 : 20
- scénario 4 : 30

Pour l'expérimentation la qualité des actionneurs a aussi été choisie arbitrairement. L'hexapode est considéré de meilleure qualité que l'AIBO car il a plus de batterie.

Les deux scénarios de pilotage permettent de tester la partie Actionneur générique de l'algorithme. En effet voici les différent cas observés.

- Cas 1 : Le participant A pilote l'AIBO avec le smartphone. Le participant B pilote un robot avec la manette. Résultat : Le participant B pilote l'hexapode.

- Cas 2 : Le participant A pilote l'hexapode avec le HTC. Le participant B pilote un robot avec la manette. Résultat : Le participant B pilote l'AIBO.
- Cas 3 : Pas de participant A. Le participant B pilote un robot avec la manette. Résultat: Le participant B pilote l'hexapode.
- Cas 4 : Le participant B pilote un robot avec la manette. Le participant B pilote l'hexapode. Le participant A arrivé et choisi de piloter l'hexapode avec le HTC. Résultat: Le participant A pilote l'hexapode. Le participant B se met à piloter l'AIBO.

Le scénario 4 qui est le plus prioritaire permet de tester la synchronisation d'action. Durant l'expérimentation lorsque l'on mettait le code barre correspondant les trois actionneurs commencaient leur action en même temps. Le scénario 3 a permis de tester la libération des actionneurs. En effet il arrivait souvent que la camera détecte une tête pendent l'action synchronisé. Cependant GRETA exécutait sont action de beaviorHello() que lorsque l'AIBO avait fini son salue car c'est lui qui mettait le plus de temps.

L'objectif de gestions des conflits était testé par l'ensemble des scénarios. Pendant l'expérience aucun actionneur n'a reçu deux ordres en même temps.

L'expérience a permis de voir que le gestionnaire se comportait correctement dans les cas testés ci-dessous. Par la suite il serait cependant intéressant de tester avec un plus grand nombre d'actionneur et plus de scénario en parallèle. La réactivité des actionneurs était parfois un peu lente, cela peut venir du gestionnaire mais aussi de l'état des différentes connexions (wifi principalement) ainsi que la vitesse des interpréteurs. Il serait donc intéressant de mesurer les temps de calculs du gestionnaire.

5 CONCLUSION

Les environnements, dans lesquels nous évoluons, sont composés de plus en plus de dispositifs électroniques. L'intelligence ambiant a pour but de faciliter et de rende plus intuitive notre interaction avec ces dispositifs. Cela peut passer par une automatisation de ces dispositifs uu encore en communiquant avec eux de manière plus naturelle, par exemple avec la voix ou les gestes.

Dans le cadre du projet Robadom une architecture a été développée afin de permettre aux utilisateurs de programmer leur environnement. Ils créent un ensemble de scénario qui s'exécutent en parallèle et qui utilisent des capteurs et des actionneurs de l'environnent.

Le module de gestion de monde que j'ai développé permet de gérer les conflits qui peuvent subvenir entre plusieurs scénarios. Il permet aussi deux nouvelles fonctionnalités de l'architecture : le choix d'un actionneur générique au moment de la création des scénarios et la possibilité de synchroniser un ensemble d'actionneurs.

Des améliorations sont envisagées pour la fin du stage (fin juillet), afin de perfectionner les fonctionnalités existantes ou d'en offrir de nouvelles. La première amélioration est la prise en compte des connexions et déconnexion des actionneurs. En effet pour le moment si un scénario envoi un ordre à un actionneur qui est déconnecté celui-ci est ignoré. Une seconde amélioration serait de permettre à la parallélisation synchronisée de contenir plusieurs actions en séquence par branche.

Références bibliographiques :

[1] Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., And Burgelman, J.-C. 2001. Scenarios for ambient intelligence in 2010. IST Advisory Group Final Report, European Commission, EC. Brussels.

[2] Sadri, F. 2011. Ambient Intelligence: A Survey. Acm Computing Surveys, 43(4).

[3] Garate, A.,Herrasti, N., and Lopez, A. 2005. GENIO: An ambient intelligence application in home automation and entertainment environment. In Proceedings of the Joint sOc-EUSAI Conference. 241–245.

[4] Cook, D. J.,Youngblood, M., And Das, S. K. 2006. Amulti-agent approach to controlling a smart environment. In Designing Smart, The Role of Artificial Intelligence, J. C Augusto and C.D. Nugent, Eds., Lecture Notes in Artificial Intelligence, vol. 4008, Springer, Berlin, 165–206.

[5] Hagras, H., Callaghan, V., Colley, M., Clarke, G., Pounds-Cornish, A., And Duman, H. 2004. Creating an ambient-intelligence environment using embedded agents. IEEE Intell. Syst. 12–20.

[6] Bahadori, S., Cesta, A., Iocchi, L., Leone, G. R., Nardi, D., Pecora, F., Rasconi, R., And Scozzafava, L. 2004a. Towards ambient intelligence for the domestic care of the elderly, Robocare Tech. Rep. RC-TR-0704-3.

[7] Page, A. J. and Naughton, Th. J. Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. 8th International Workshop on Nature Inspired Distributed Computing. In: Proceedings of the 19th International Parallel & Distributed Processing Symposium, Denver, Colorado, USA, IEEE Computer Society, April 2005.

[8] C. Wang , H. Ghenniwa and W. Shen 2008 Distributed scheduling for reactive maintenance of complex systems, IEEE Trans. Syst., Man, Cybern. C, Appl. Rev., vol. 38, pp.269

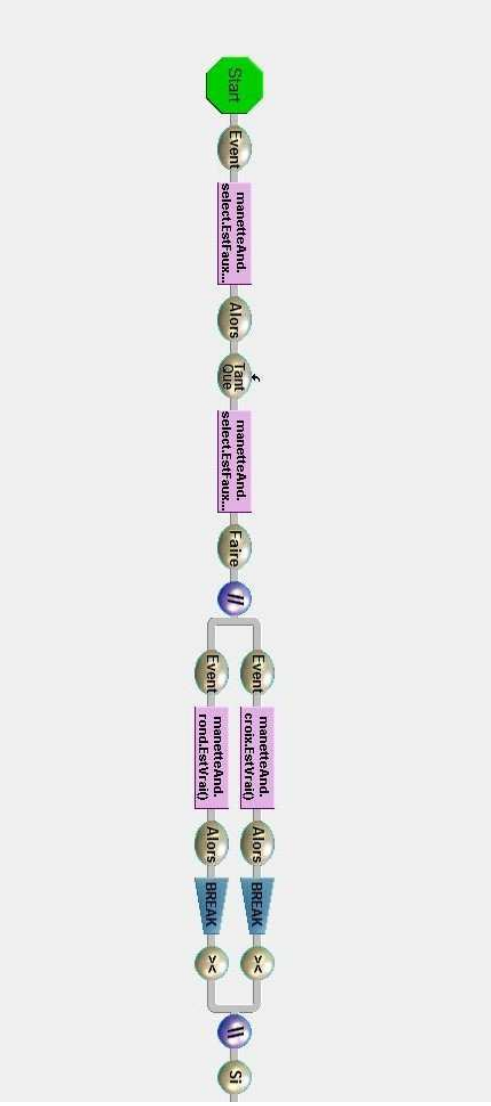
[9] R. G. Smith 1980 "The contract net protocol: High-level communication and control in a distributed problem solver", IEEE Trans. Comput., vol. C-29, pp.1104

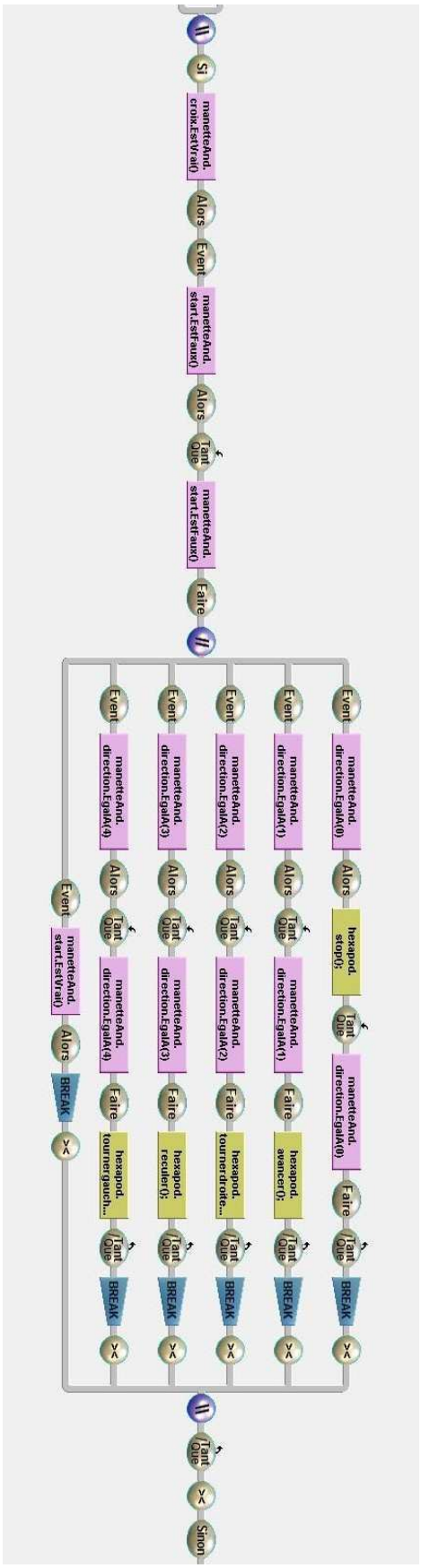
[10] A. Madureira, N. Sousa, I. Pereira 2011 Negotiation mechanism for self-organized scheduling system, Third World Congress on Nature and Biologically Inspired Computing (NaBIC) , pp. 291–296

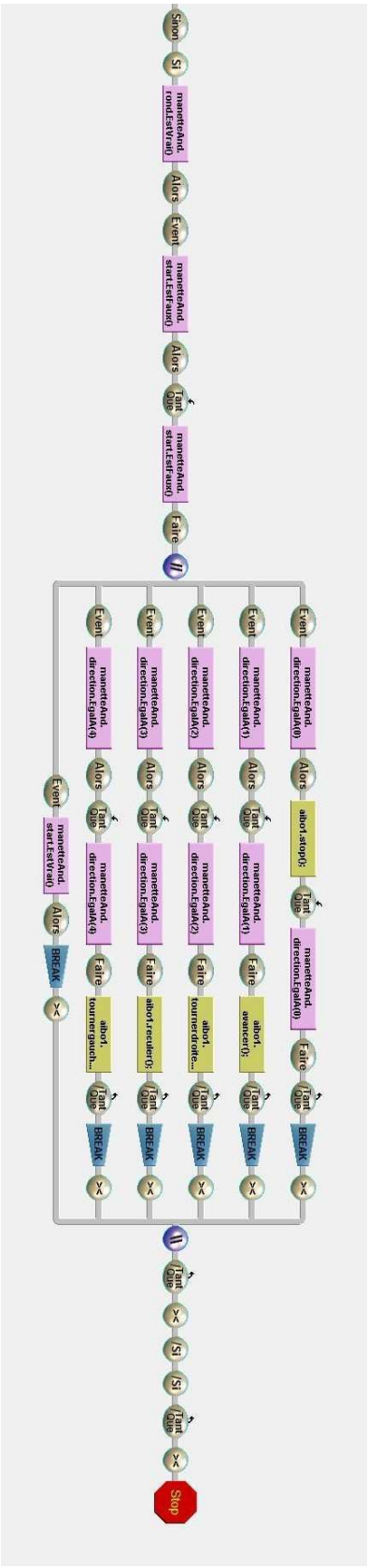
6 ANNEXES

6.1 SCENARIOS

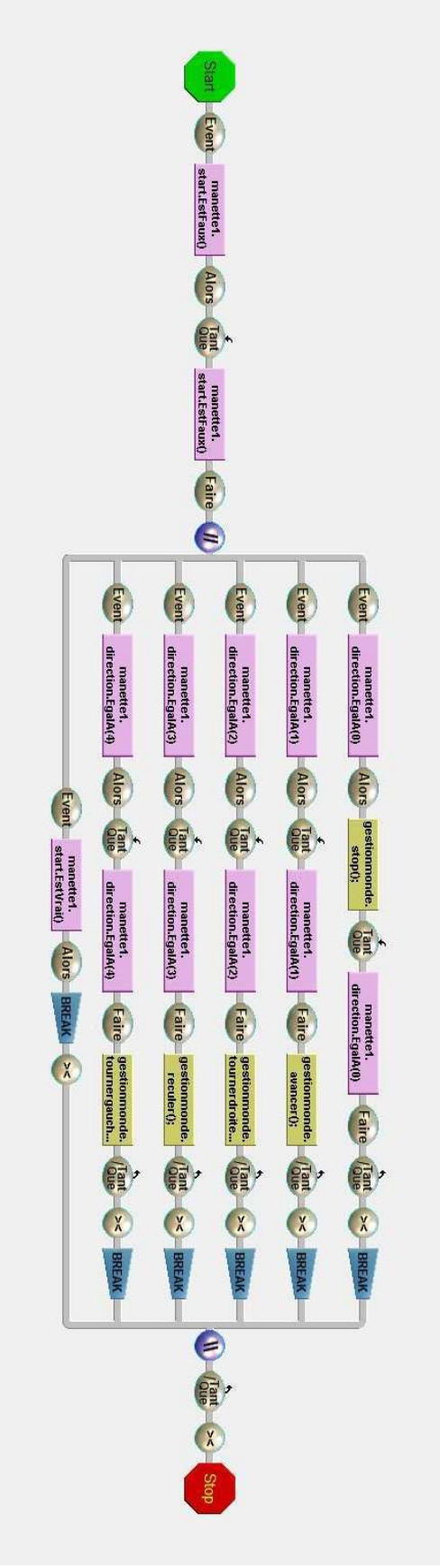
6.1.1 PILOTAGE ROBOT SPECIFIQUE


Start
Event
manetteAnd.
select.EstFaux...
Alors
Tant
que
manetteAnd.
select.EstFaux...
Faire
//
Event
manetteAnd.
rond.EstVrai()
Alors
BREAK
><
Event
manetteAnd.
croix.EstVrai()
Alors
BREAK
><
//
Si


//
Si
manetteAnd.croix.EstVrai()
Alors
Event
manetteAnd.start.EstFaux()
Alors
Tant Que
manetteAnd.start.EstFaux()
Faire
//
Event
manetteAnd.direction.EgalA(0)
Alors
hexapod.stop();
Tant Que
manetteAnd.direction.EgalA(0)
Faire
Tant Que
BREAK
><
Event
manetteAnd.direction.EgalA(1)
Alors
Tant Que
manetteAnd.direction.EgalA(1)
Faire
hexapod.avancer();
Tant Que
BREAK
><
Event
manetteAnd.direction.EgalA(2)
Alors
Tant Que
manetteAnd.direction.EgalA(2)
Faire
hexapod.tournerdroite...
Tant Que
BREAK
><
Event
manetteAnd.direction.EgalA(3)
Alors
Tant Que
manetteAnd.direction.EgalA(3)
Faire
hexapod.reculer();
Tant Que
BREAK
><
Event
manetteAnd.direction.EgalA(4)
Alors
Tant Que
manetteAnd.direction.EgalA(4)
Faire
hexapod.tournergauch...
Tant Que
BREAK
><
Event
manetteAnd.start.EstVrai()
Alors
BREAK
><
//
Tant Que
><
Sinon



6.1.2 *PILOTAGE ROBOT GENERIQUE*


Start
Event
manette1.start.EstFaux()
Alors
Tant Que
manette1.start.EstFaux()
Faire
//
Event
manette1.direction.EgalA(0)
Alors
gestionmonde.stop();
Tant Que
manette1.direction.EgalA(0)
Faire
Tant Que
><
BREAK
Event
manette1.direction.EgalA(1)
Alors
Tant Que
manette1.direction.EgalA(1)
Faire
gestionmonde.avancer();
Tant Que
><
BREAK
Event
manette1.direction.EgalA(2)
Alors
Tant Que
manette1.direction.EgalA(2)
Faire
gestionmonde.tournerdroite...
Tant Que
><
BREAK
Event
manette1.direction.EgalA(3)
Alors
Tant Que
manette1.direction.EgalA(3)
Faire
gestionmonde.reculer();
Tant Que
><
BREAK
Event
manette1.direction.EgalA(4)
Alors
Tant Que
manette1.direction.EgalA(4)
Faire
gestionmonde.tournergauch...
Tant Que
><
BREAK
Event
manette1.start.EstVrai()
Alors
BREAK
><
//
Tant Que
><
Stop

6.1.3 BONJOUR GRETA

6.1.4 SYNCHRONISATION