Ministère de l'Education Nationale

Université de Montpellier II

Rapport de Projet Informatique FMIN200
Du Master informatique 1ere année

Réalisation d'un langage graphique pour Warbot

Etudiants :
AYITE Adama Karim
Bouraoui Nidhal
Kambie Mouhamadou
Kchir Selma

Encadrant :
Ferber Jacques

Remerciements

Nous tenons à remercier monsieur Ferber pour sa disponibilité, ses directives et ses conseils judicieux ainsi qu' à tout le cadre enseignant du département informatique de l'université Montpellier 2.

INTRODUCTION

La programmation orientée agent est un domaine qui prend de plus en plus d'importance dans la gestion de logiciel amenés à interagir entre eux ou avec leur environnement. Elle est utilisée par exemple dans la domotique ou encore dans les jeux vidéos, et peut même être utilisée pour modéliser le comportement d'animaux. C'est cette dernière qualité qui nous intéresse dans le cadre du projet que nous devons réaliser. En effet, il nous a été proposé de réaliser un langage graphique permettant de définir le comportement des robots du jeu de stratégie Warbot. Actuellement, le comportement des warbots doit être programmé à la main, ce qui demande un minimum de connaissances en programmation et notamment la programmation orienté objet. Nous devons donc faciliter l'accès du jeu de stratégie Warbot en permettant à un utilisateur non informaticien de pouvoir définir les actions que ses robots doivent effectuer sans avoir à taper du code lui-même. Le moyen le plus simple de le faire est de fournir à l'utilisateur une interface graphique lui permettant de s'abstraire des langages de programmation, cela implique qu'il faut trouver un moyen de représentation visuel et formel du comportement du robot. La théorie des automates nous fournit un outil parfaitement adapté pour la modélisation des comportements, la machine à états finis. Cette dernière fournira la base des informations de description du comportement qui sera facilement compréhensible pour l'utilisateur novice, autrement dit la machine sera à la base de notre langage graphique. De notre côté, il nous sera facile par la suite de créer du code dans un langage de programmation, dans notre cas python et java, qui sera exploitable par Warbot. Cela amènera à Warbot un public plus large et suscitera peut-être même des vocations pour la programmation orientée agent.

I. Spécification des besoins et Modélisation

1. Spécification des besoins

A. Cahier des charges

Le travail qui nous a été confié consiste à concevoir un plugin qui présente un langage graphique permettant de définir des comportements d'agents à travers une machine à états finis.

- Il s'agit de faciliter la personnalisation des comportements d'agents à partir d'un éditeur graphique de comportement.

- A partir de cette représentation graphique, on devrait pouvoir générer un code en Python ou en Java décrivant une stratégie donnée pour un agent donné.

- Cette représentation graphique est un formalisme sous Sedit que nous aurons à implémenter et intégrer sous MadKit.

- Le code généré devrait s'intégrer dans un projet Warbot permettant aux agents d'avoir le comportement souhaité.

- Le code généré devra être en anglais (commentaires, déclaration de variables, etc...)

B. Etude l'existant

MadKit est une plate forme de développement de systèmes multi-agents fonctionnant en java destinée au développement et à l'exécution de systèmes multi-agents et plus particulièrement à des systèmes multi-agents fondés sur des critères organisationnels (groupes et rôles).

MadKit n'impose aucune architecture particulière aux agents. Il est

ainsi possible de développer aussi bien des applications avec des agents réactifs comme le fait TurtleKit que des agents cognitifs et communicationnels, et même de faire interagir aisément tous ces types d'agents.

Cela permet ainsi aux développeurs d'implémenter l'architecture de leur choix.

MadKit est écrit en Java et fonctionne en mode distribué de manière transparente à partir
d'une architecture "peer to peer" sans nécessité de serveur dédié. Il est ainsi possible de faire communiquer des agents à distance sans avoir à se préoccuper des problèmes de communication qui sont gérés par la plate-forme.

MadKit est construit autour du concept de "micro-noyau" et "d'agentification de services". Le
Micro-noyau de MadKit est très petit moins de 100Ko de code, car il ne gère que les organisations (groupes et rôles) et les communications à l'intérieur de la plate-forme. Le mécanisme de distribution, les outils de développement et de surveillance des agents, les outils de visualisation et de présentation sont des outils supplémentaires qui viennent sous la forme d'agents que l'on peut ajouter au noyau de base.

De ce fait, MadKit peut être utilisé aussi bien comme outil de développement d'applications que comme un noyau d'exécution de système multi-agent qui peut être embarqué dans des applications quelconques.

Madkit contient plusieurs plate-formes de développement, dont notamment Warbot, un logiciel de simulation de combat et contient aussi SEdit, qui est un logiciel permettant de créer des formalismes graphiques.

Plugin

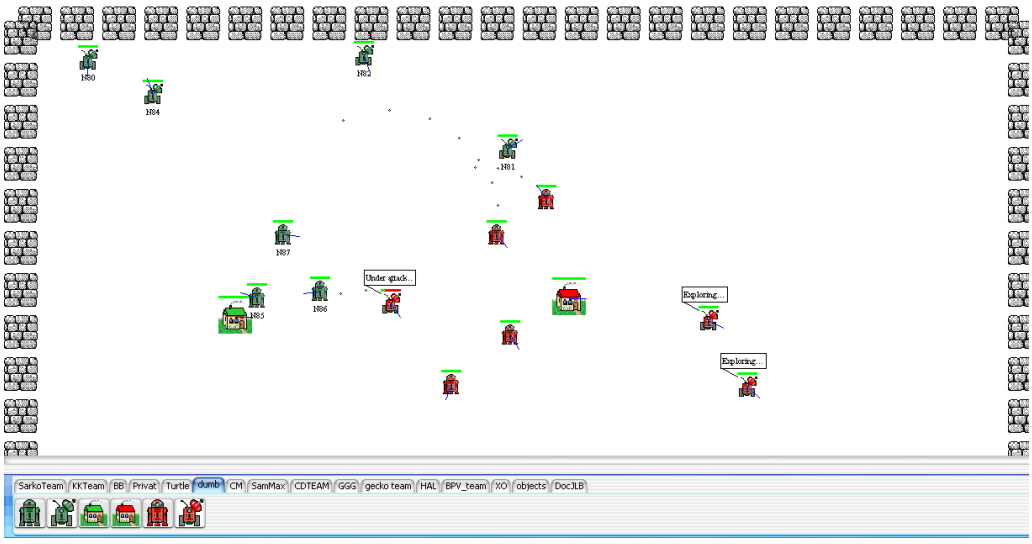
Un plugin est un logiciel qui complète un logiciel hôte pour lui apporter de nouvelles fonctionnalités.

La plupart du temps, les plugins sont caractérisés de la façon suivante :

- ils ne peuvent fonctionner seuls car ils sont uniquement destinés à apporter une fonctionnalité à un ou plusieurs logiciels ;
- ils sont mis au point par des personnes n'ayant pas nécessairement de relation avec les auteurs du logiciel principal.

Warbot

Warbot est donc un plugin de MadKit. Chaque "robot" dans Warbot est divisé en deux: une partie 'corps' qui est prédéfinie et qu'on ne peut pas modifier, et une partie 'tête' qu'il s'agit de modifier pour que son équipe de robots gagne. A l'heure actuelle, il faut coder le comportement de ces agents, soit en python, soit en java.



SEdit

Sedit est un plugin intégré à MadKit qui permet de visualiser et d'animer des diagrammes de type "automates", "portes logiques" ou "réseaux de petri".Ce plugin fonctionne essentiellement à base de formalismes.

Formalisme

Un formalisme est un fichier xml qui contient plusieurs informations permettant la création d'une fenêtre SEdit. Ces informations sont pour la plupart des classes java, préalablement compilées et ajoutées à la librairie MadKit.

2. Modélisation

A. Description de la machine à états finis

Pour satisfaire les nécessités du plugin nous avons conçu une machine à états finis constituée d'états décrivant les activités pour chaque agent et de transitions qui contiennent les conditions selon lesquelles un agent passe d'un état à un autre. Pour pouvoir assurer la compatibilité du plugin avec le reste des modules nous avons opté pour la génération d'un fichier XML qui contient une description de la machine à état sous forme de balises XML dont la grammaire est décrite ci-dessous.

i) DTD associés de la machine à états finis

<! ELEMENT FSM (State+)>
<! ATTLIST FSM firststate CDDATA #REQUIRED>
<! ATTLIST FSM robotType CDDATA #REQUIRED>
<! ELEMENT State (transition*)>
<! ATTLIST transition next-state CDDATA #REQUIRED>
<! ATTLIST priority next-state CDDATA #REQUIRED>
<! ELEMENT transition (not-condition?, and-condition| or-condition| nand-condition| nor-condition)>
<! ELEMENT and-condition (not-condition?, and-condition| or-condition| nand-condition| nor-condition)>
<! ELEMENT or-condition (not-condition?, and-condition| or-condition| nand-condition| nor-condition)>
<! ELEMENT nand-condition (not-condition?, and-condition| or-condition| nand-condition| nor-condition)>
<! ELEMENT nor-condition (not-condition?, and-condition| or-condition| nand-condition| nor-condition)>

ii) Exemple de fichier de la machine

<FSM firststate="l'état initial de la FSM" robotType = "le type du robot">
<State name="nom d'état de départ" activity="activité associée à cet état">
<transition next-state="nom de l'état d'arrivée" priority="la priorité de la transition">
<and-condition>
une condition
<not-condition> une autre condition </not-condition>
</and-condition>
</transition>
<transition next-state="nom d'un autre état d'arrivée" priority="la priorité de la transition différente de la première">
<and-condition>
une condition
<not-condition> une autre condition </not-condition>
</and-condition>
</transition>
</State >
</FSM>

B. Les états

Un état est identifié par un nom et une activité. La nécessité de séparer les notions d'état et d'activité est due au fait qu'un agent pourrait avoir deux états différents contenant la même activité mais de transitions différentes. Un état élément de la machine à états finis dont les balises sont décrites précédemment. Il est caractérisé par l'ensemble des transitions à qui il réfère et par les attributs firststate et robotType qui indiquent respectivement si l'état est un état d'initialisation et le type du robot en question.

C. Les activités

Une activité d'un agent c'est un comportement de base qui regroupe un ensemble d'actions propres à un agent donné sous Warbot. Ces actions sont disponibles sur l'api de Warbot. Chaque activité est spécifique à un robot donné et classée dans une catégorie facilitant la recherche et le tri des activités selon les catégories. Ces données sont décrites dans un fichier XML et sont chargées automatiquement au lancement du plugin. Cela assurerait la réutilisabilité de l'application et rendrait possible l'ajout et la modification des activités sans avoir à recompiler le code du plugin. Chaque activité est une balise qui décrit le nom de cette dernière, le type d'agent qui peut l'exécuter, la catégorie à laquelle elle appartient, le langage avec lequel elle est écrite ainsi qu'un chemin vers le code source de cette dernière.

Prenons par exemple l'activité Seek-food, le robot n'effectue cette activité que si certaines conditions sont vérifiées suivant la machine à états finis, le nom de l'activité correspondant à cet état est donc appelé grâce à l'attribut name et son code source contenu dans la classe Java(le langage est indiqué dans l'atribut langage de la balise XML) dont le chemin est indiqué dans l'attribut path est exécuté.

```
<Activity name="Seek-food" agent="RocketLauncher" category="Eat"
language="Java" path="" desc="" />
```

Les activités diffèrent d'un type d'agent à un autre, par exemple les robots explorateurs n'attaquent pas les ennemis, la base convertit la nourriture en robots, les explorateurs envoient des messages pour prévenir son équipe de la présence d'un ennemi... nous avons donc défini un attribut agent afin de distinguer le comportement des agents.

i) DTD associés au fichier XML des activités

```
<! ELEMENT Activities (Activity +)>
<! ATTLIST Activity name CDDATA #REQUIRED>
<! ATTLIST Activity agent CDDATA #REQUIRED>
<! ATTLIST Activity category CDDATA #REQUIRED>
<! ATTLIST Activity language CDDATA #REQUIRED>
<! ATTLIST Activity path CDDATA #REQUIRED>
<! ATTLIST Activity desc CDDATA #REQUIRED>
```

ii) Exemple de fichier XML d'activités

```
<Activities>
 <Activity name="patrouille aléatoire " agent="RocketLauncher"
 category=" patrouilles " language="python" path="patrouiller.py"
 desc="Make robot random - patrol" />
<Activity name="attaquer l'ennemi le plus proche"
agent="RocketLauncher" category="Attaques" language="python"
path="attaquer.py" desc="Make robot attack the first ennemy" />
```

D. Les transitions

Une transition est identifiée par un état de départ et un état d'arrivée. Elle englobe l'ensemble des conditions de bases sous forme d'arbre définissant une expression bien formée constituée d'opérateurs logiques et de conditions.

Ainsi, pour décrire par exemple l'expression « A ⋀ (¬B V C) ⋀¬D », obtient dans la balise transition du fichier XML la structure suivante :

```
<and-condition>
        A
        <or-condition>
                <not-condition>
                        B
                <not-condition>
                        C
        </or-condition>
<not-condition>
        D
<not-condition>
</and-condition>
```

E. Les conditions

Une condition devrait comporter les différents indices pour chaque robot ainsi une évaluation de ce dernier. L'ensemble des ces indices constitue l'état interne et l'environnement d'un agent à un temps donné.

F. Les indices

Un indice est une méthode développée en java ou en python qui décrit un percept donné (existence d'un ennemi) ou un état d'un agent (niveau d'énergie). On distingue plusieurs types de percepts : la nourriture, les missiles, le mur, les ennemis...

Le comportement des agents est défini selon les percepts détectés dans un rayon de perception donné.

Par analogie avec les activités, les indices sont aussi stockés dans un fichier XML permettant la gestion dynamique de ces derniers. Chaque indice est une balise qui décrit le nom de ce dernier, le langage avec lequel il est écrit ainsi qu'un chemin vers le code source de ce dernier.

i) DTD associés au fichier XML des indices

```
<! ELEMENT Indices (Indice +)>
<! ATTLIST Indice name CDDATA #REQUIRED>
<! ATTLIST Indice language CDDATA #REQUIRED>
<! ATTLIST Indice path CDDATA #REQUIRED>
```

ii) Exemple de fichier XML des indices

```
<?xml version="1.0" encoding="UTF-8"?>
<Indices>
< Indice name="nombre ennemies" language="Java"
path="./ressources/ennemies/ennemiesnbre.java "/>
  <Indice Indice name="nombre d'alliés" language="Java"
path="./ressources/ennemies/allynbre.java"/>
  <Indice name="nourriture" language="Java" path="ressources/ennemies/
eat.java"/>
  <Indice name="energy" language="Python"
path="./ressources/energy.java"/>
</Indices>
```

G. Génération de code

Le code xml résultant de la machine à états finis est un code générique indépendant du langage avec lequel les indices et les activités sont développés. C'est une sorte de langage intermédiaire contenant des états et des transitions vers d'autres états. Pour ces raisons là, on a opté pour un schéma « Pattern State » qui convient parfaitement à nos besoins. Ainsi pour une classe d'état donnée du pattern State, on associe le code d'une activité donnée et pour chaque transition on associe une méthode qui permet de changer vers l'état souhaité et pour chaque indice on associe la méthode décrite dans le fichier source dont le chemin est contenu dans le fichier XML des indices. Ce qui nous permettrait la conversion du code XML en code java ou en code Python.

II. Déploiement de l'application sous Sedit

Avant toute intégration sous SEdit,nous avons besoin d'éditer un fichier XML qui se trouve être le *formalisme ,* et le *plugin* ***Brain*** dans lequel se trouve le code d'implémentation des différents éléments du formalisme .

1. Formalisme de Brain: *Brain.fml*

Le formalisme est donc la définition des nœuds de l'application sous Sedit. Tous les nœuds sont connectés par un élément de liaison. Un formalisme SEdit est un fichier formaté XML.

Dans la conception que nous avons faite et qui est décrite plus haut, nous avons eu besoin que d'un type de nœuds: les états et à cela s'ajoutent les liaisons entre eux.

La bonne marche d'un formalisme nécessite d'indiquer dans *Brain.fml* les classes d'implémentation de la structure, des nœuds, des liaisons. Grâce au formalisme, en plus de pouvoir définir très simplement des actions sur des nœuds, sur des liaisons mais aussi sur la structure,nous pouvons ajouter tous les boutons dans la barre d'outils de SEdit et en gérer les événements.

Nous avons défini un formalisme tout simple dont voici les grandes lignes:

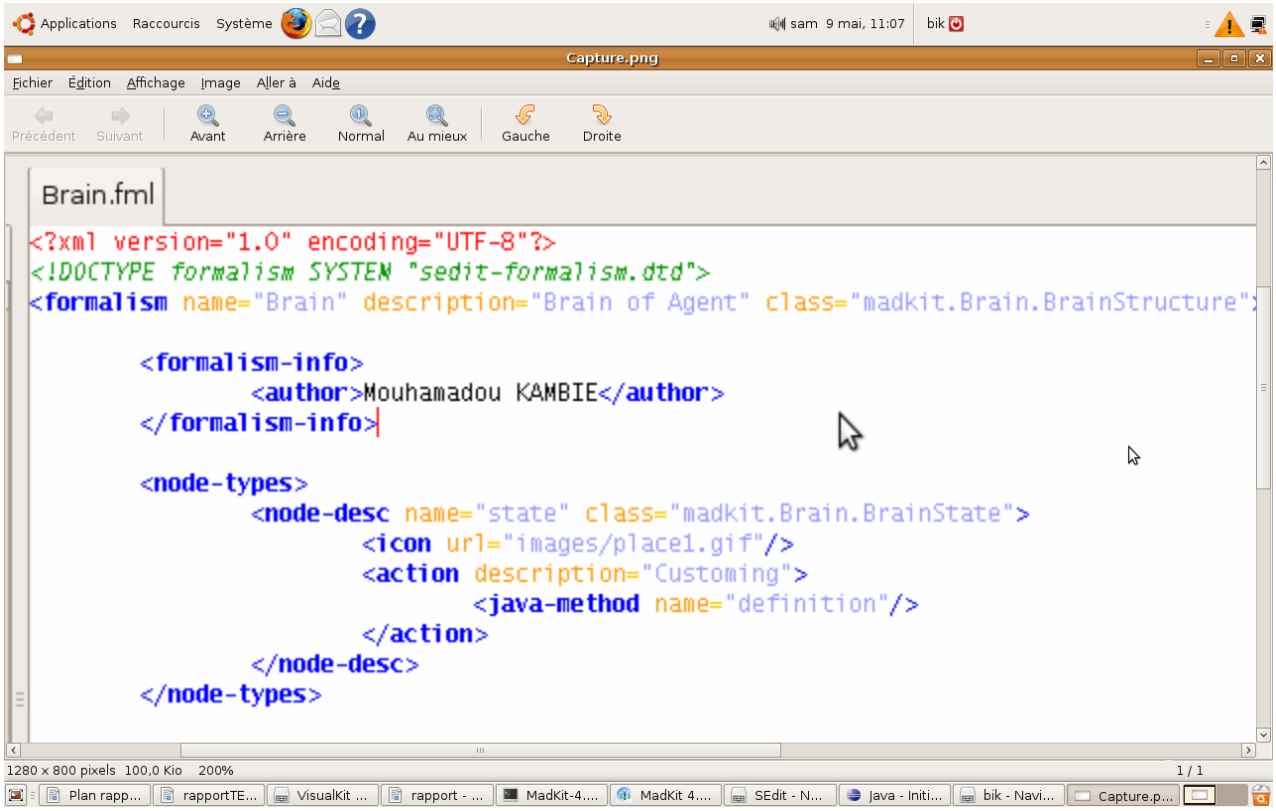


fig4.1

Ici nous avons le début de notre formalisme. Les attributs de la balise racine: *formalism* permettent entre autre d'indiquer le nom de la classe principale d'implémentation dans l'attribut *class* ;nous y reviendrons plutard.Les balises formalism-info étant des balise d'information des auteurs entre autres.

Ensuite il faut définir les types de nœuds ainsi que leur classe d'implémentation : Nous en avons qu'un seul symbolisé par la balise *node-desc* et qui représente l'état; sa visualisation graphique étant réalisée grâce à la balise *icon.*

En outre, l'ajout d'une action sur un noeud de SEdit,se fait au moyen de la balise *description,* ici de l'état courant. Ainsi un clic droit sur ce dernier fait apparaître un menuPopup dans lequel figure la valeur de la l'attribut *name* de cette balise(*description*). L'événement associé au clic est géré dans la classe du nœud courant par la fonction de l'attribut *name* de la balise *java-method.* Dans l'exemple de la figure *fig 4.1,* il s agit de la fonction definition qui caractérise ou spécifie l'état courant; nous y

reviendrons plus en détails.

La figure *fig 4.2* donne la suite du formalisme

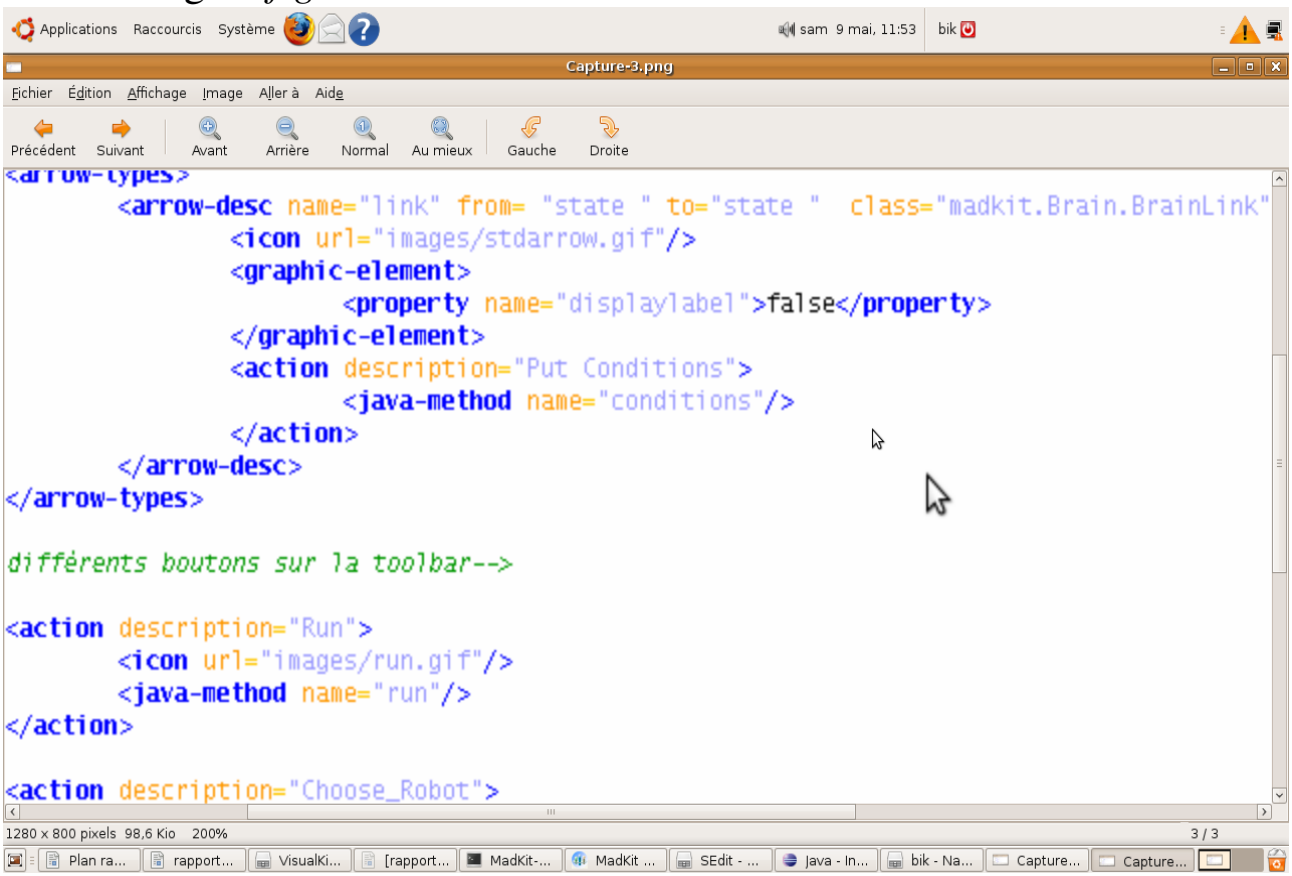


fig 4.2

Nous remarquons ici les balises *arrow-types* et *action*.

La balise *arrow-types* définit les liaisons. Ainsi, sa balise imbriquée *arrow-desc*, tout comme *node-desc* vu précédemment, attribue un nom à la dite liaison et lui affecte sa classe d'implémentation; ici *madkit.Brain.BrainLink*. La balise *graphic-element* autorise ou non l'édition d'un nom sur la liaison.La balise *description* ici permet à l'utilisateur d'ajouter des conditions sur cette liaison.

La balise *action* enfin ajoute des boutons à la barre d'outils de SEdit et lui associe des événements à l'aide de la même balise *java-method.* Dans l'exemple d'en haut il s'agit du bouton *run* dont le gestionnaire d'événements run de la balise *java-method* est implémentée dans la classe principale de du formalisme: *madKit.Brain.Structure.*

Lors de la création d'un nœud, l'instance de la classe de la structure ajoute une nouvelle instance de la classe définissant le nœud, SEdit dessine ensuite l'élément suivant les instructions de cette dernière.

2. Implémentation du plugin Brain

N'ayant pas eu besoin des noeuds transitions, il s'agit alors d' implémenter les classes ***BrainStructure,BrainState,BrainLink*** implémentant respectivement la structure principale de l'application SEdit, les états sous SEdit et les liaisons.

On pourra les représenter graphiquement comme suit:

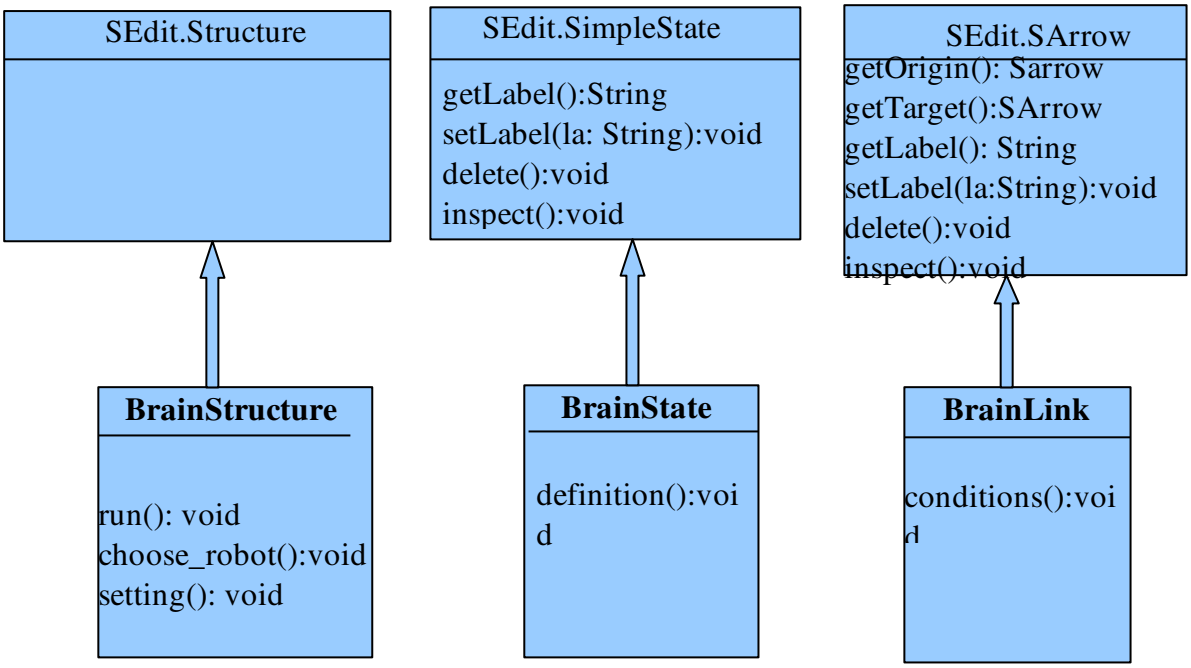


fig 4.3

A. La classe BrainState

Comme le montre la figure 4.3, elle hérite de la classe SimpleState définie dans SEdit. C'est elle qui implémente le code correspondant aux différents noeuds états sous SEdit.

Les fonctions spécifiées dans la classe mère sont celles dont nous avons besoin; cependant il y'a d'autres qui sont prédéfinies.

-La fonction *getLabel* nous retourne alors le nom de l'état courant

-La fonction s*etLabel* affecte un nom à l'état courant

-Les fonctions delete et inspect permettent respectivement de supprimer et de suivre voire inspecter l'état courant.

La fonction *definition* implémentée dans ***BrainState*** spécifie toutes les caractèristiques de l'état courant. Ainsi un click droit sur un état et une selection du menuItem Customing génère l'interface de définition suivante:

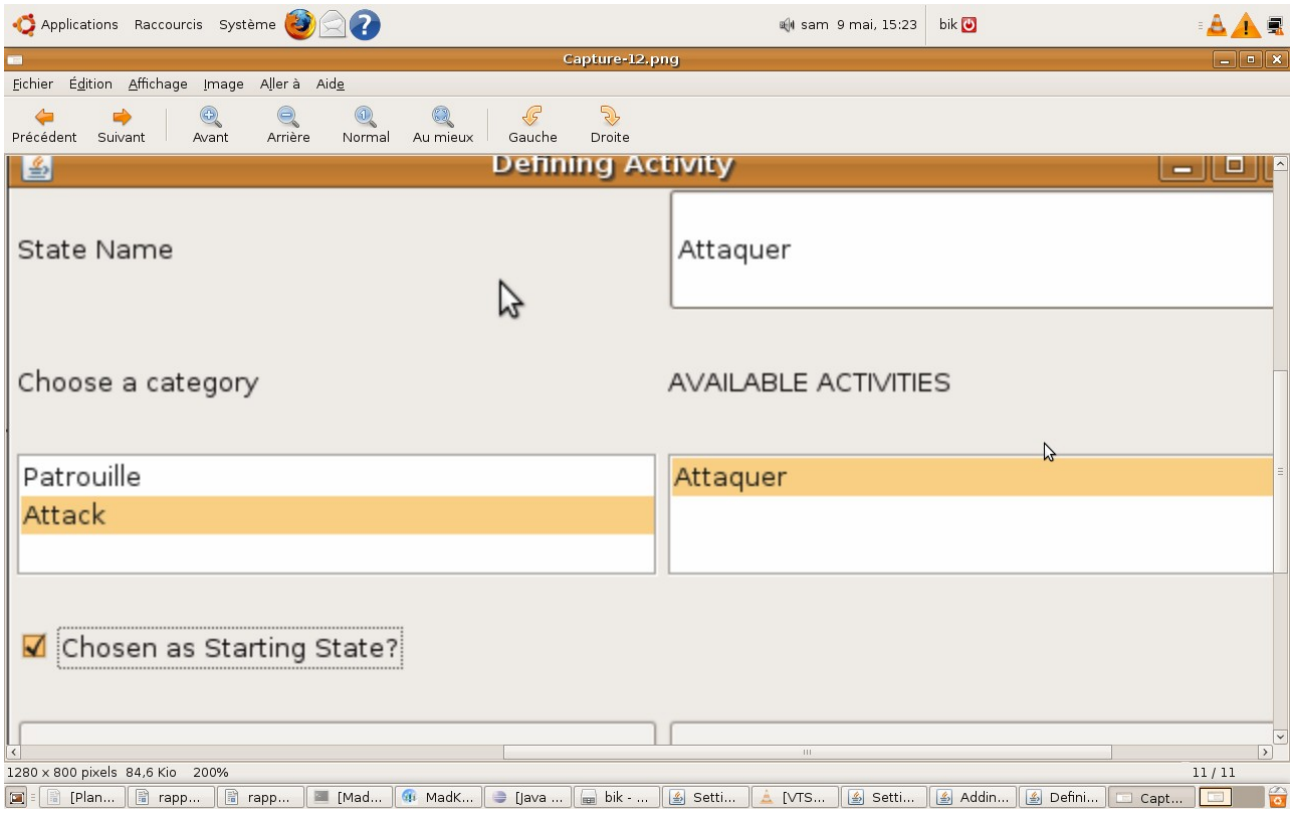


fig 4.4

Evénements associés au bouton « validate »

Rappelons d'abord que la liste des catégories est chargée automatiquement dès lors que l'utilisateur a fini de choisir le robot à représenter.En effet nous parsons le document *Activities.xml* et récupèrons les catégories correspondant au robot choisi. Pour chaque catégorie, nous listons l'ensemble des activités disponibles.

Ainsi une fois le nom de l'état saisi, et l'activité choisie, nous allons : le récupérer, parser notre document FSM et voir s'il existe dèjà un état du même nom, auquel cas nous le signalons à l'utilisateur .Sinon si la case « Chosen as Starting State » est cochée, nous le définissons comme état de départ dans l'attribut *firstate* de la racine de notre document, ensuite nous construisons la balise « State » avec comme attribut « name »: le nom saisi, enfin nous lui associons l'activité choisie et l'ajoutons à notre arborescence.

Exemple: le fichier FSM mis à jour après appui sur le bouton « Validate » de la figure 4.4 donnera.

```
<FSM firststate= « Attaquer»>
    <State name="Patrouille" activity="Patrouiller"/>
    <State name="Fuite" activity="Fuir"/>
    <State name="Attaquer" activity="Attaquer"/>
</FSM>
```

B. La classe BrainLink

Elle hérite de la classe SArrow, mais peut aussi hériter de la classe concrète SimpleStateTransitionLink. Elle gère le code des éléments liaisons.Les fonctions nouvellement spécifiées dans la classe mère, sont:
getOrigin() : qui nous renvoie l'état de départ de cette liaison
getTarget(): qui nous renvoie l'état d'arrivée

On pourra aussi parler de leurs setteurs setOrigin(Sarrow) et setTarget();
putConditions(): cette fonction gère les transitions.

C. La classe BrainStructure

Elle hérite de la classe Structure qui se trouve dans SEdit. Cette classe est la principale de l'application SEdit et c'est elle qui instancie les différents éléments de notre formalisme à chaque appel.

Ainsi, nous avons défini à l'intérieur de cette classe les trois fonctions

- choose_robot : Elle permet tout au début à l'utilisateur, de choisir le robot ou l'agent qu'il veut représenter. Une liste d'agents chargée automatiquement à partir du fichier *Activities.xml* lui sera alors présentée et selon l'agent choisi, nous allons mettre à jour nos différents composants(comboBox,..) en ne lui chargeant que les informations liées au robot choisi.

- Setting: Cette fonction est dédiée à la configuration que l'utilisateur aurait besoin de faire en un moment donné. Il lui est alors possible de changer le type de robot utilisé, de changer le langage de description des activités de ses états, de changer l'état de départ mais aussi d'ajouter d'autres activités. Selon ses choix, nous allons mettre à jour les fichiers XML concernés par ses modifications.
  Par exemple lors de l'ajout d'une nouvelle activité, notre fichier qui répertorie l'ensemble des activités, va se voir ajouté une nouvelle . De cela découle la mise à jour de tous nos composants.

Un click de ce bouton génère alors l'interface suivante:

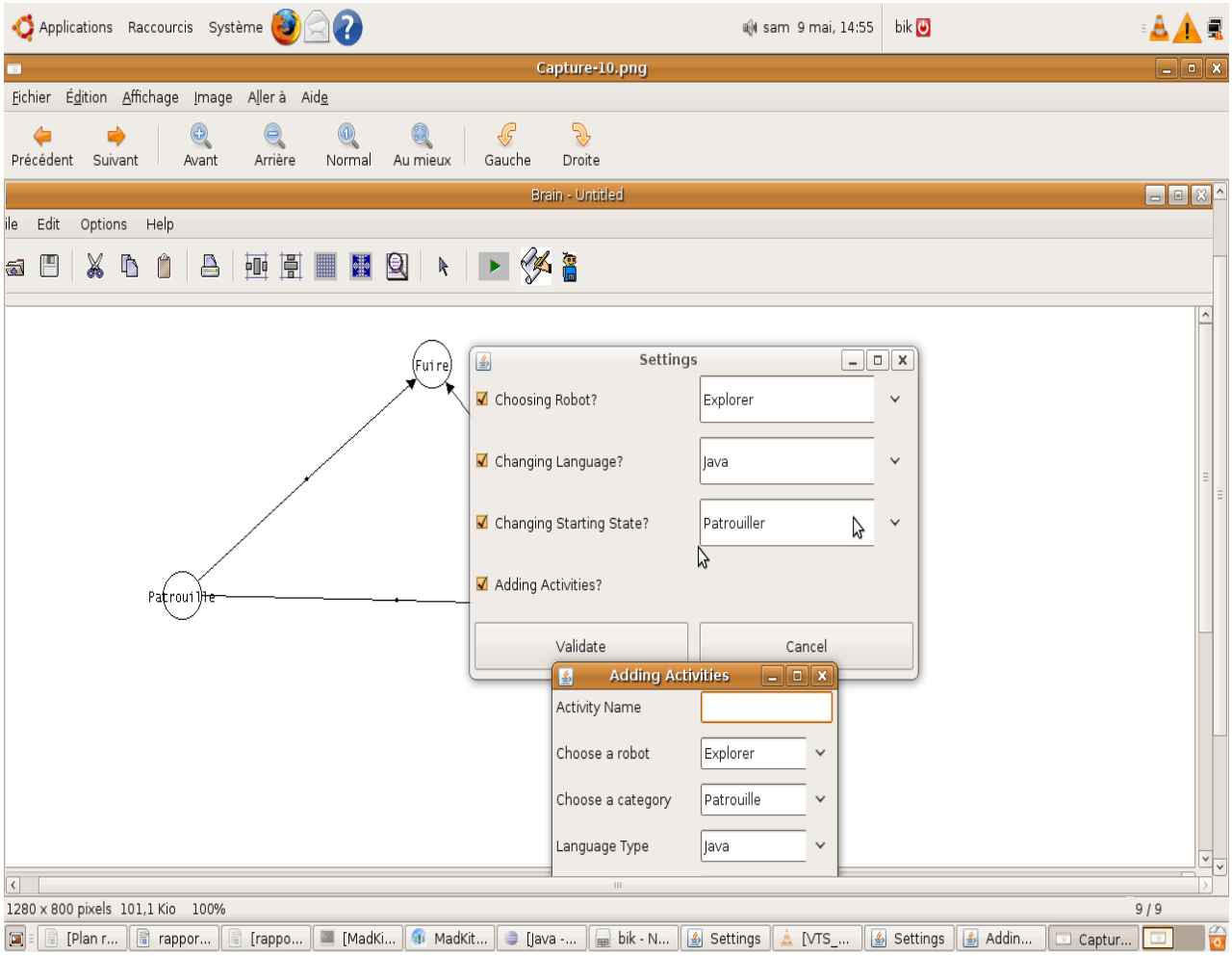


fig 4.5

- run :Cette fonction associée au bouton de même nom sur la barre d'outils de SEdit, va déclencher l'application après que l'utilisateur aura fini de définir ses états et transitions.

  Donc, nous allons récupérer toutes les informations se trouvant dans notre fichier ***FSM.xml,*** générer le code correspondant et sauvegarder celui-ci dans un fichier. Ce petit bout de code l'illustre.

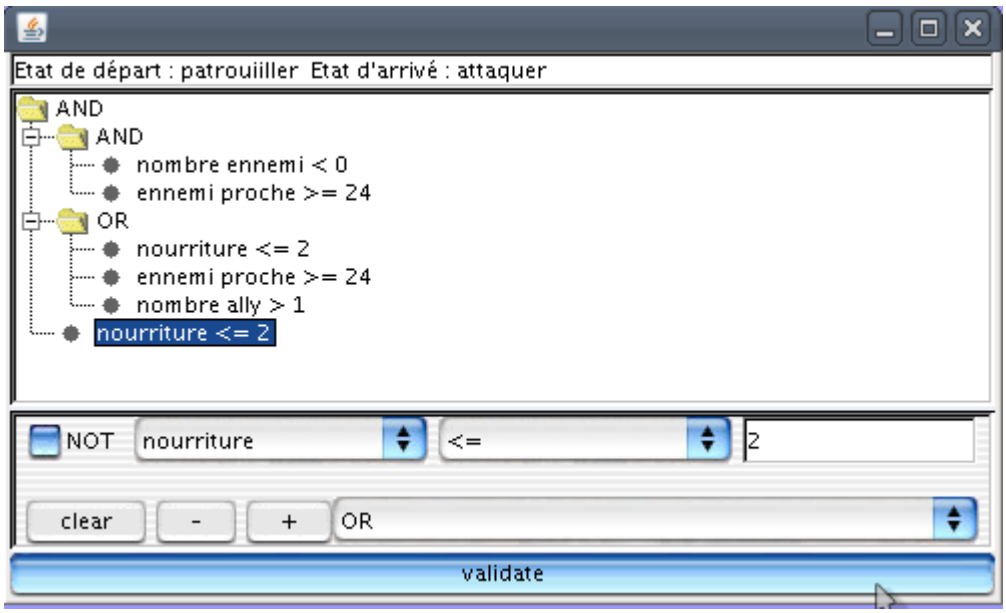
```
public void run(){
XmlToPython xp = new XmlToPython("cheminfsm",nomFichier);
            xp.writeFile();
}
```

3. Transition

Cette classe est conçue pour répondre aux besoins de configuration de chaque transition entre deux états.

A. Description de l'interface

C'est une interface d'interaction homme machine assez intuitive d'utilisation qui comporte essentiellement un arbre de conditions et un d'autres composants swing assurant la gestion (ajout, suppression..) des conditions. Le choix d'utilisation de cet arbre est imposé par la structure des conditions dans fichier XML de la machine à état finis décrite précédemment.
chaque nœud de l'arbre "JTree" correspond à un opérateur logique de base (AND,NAND,AND,OR) dont les fils sont des nœuds (opérateur logique) ou des feuilles.
Chaque feuille de l'arbre correspond à une condition ou la négation d'une condition donnée. Une condition est composé d'un indice un opérateur arithmétique et une valeur ainsi qu'un checkBox permettant d'indiquer s'il s'agit d'une négation d'une condition.



B. Exploitation des fichiers XML

Les indices contenus dans chaque condition sont chargés automatiquement à partir du fichier XML dans un « JCcomboBox » assurant la réutilisabilité de l'application et l'ajout dynamique de nouveaux indices. Au moment de la validation les données de l'arbre sont traduites sous forme de balises XML exploitable par le module de compilation pour générer du code python ou du code Java.

C. Gestion de la structure de l'arbre des Conditions

L'arbre des conditions est conçu de façon à ce qu'il puisse supporter le « drag and drop » pour changer l'ordre ou la structure des conditions on peut ainsi permettre à l'utilisateur une gestion simple et intuitive de cette dernière. Ceci est assuré en implémentant les méthodes des interfaces DragSourceListener DragGestureListener et DragTargetListener.

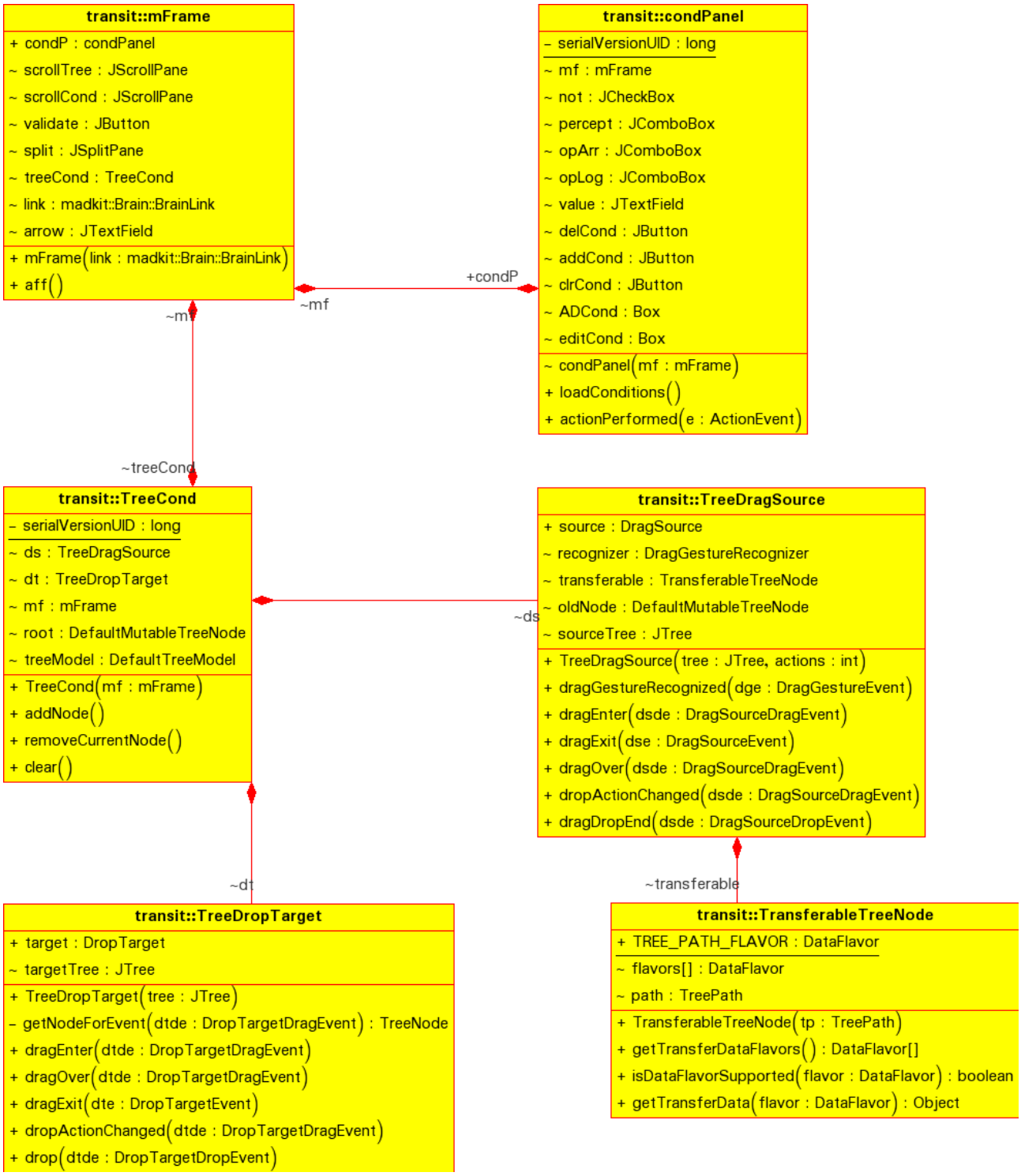
D. Intégration dans Sedit:

La classe transition comprend un objet de type SArrow grâce auquel elle peut charger dynamiquement et en temps réel le nom de l'état source et le nom de l'état destination. Lequel objet est passé au constructeur au moment de l'instanciation de la classe transition dans la classe BrainLink qui hérite à son tour de la classe SArrow.

E. Vérification syntaxique de l'arbre des conditions

Avant la génération du code intermédiaire une vérification syntaxique est effectuée sur l'ensemble des nœuds de l'arbre pour éviter une incohérence syntaxique de l'expression bien formée. Un message d'erreur est prévu pour indiquer la nature et la position de l'erreur dans l'arbre permettant ainsi une correction.

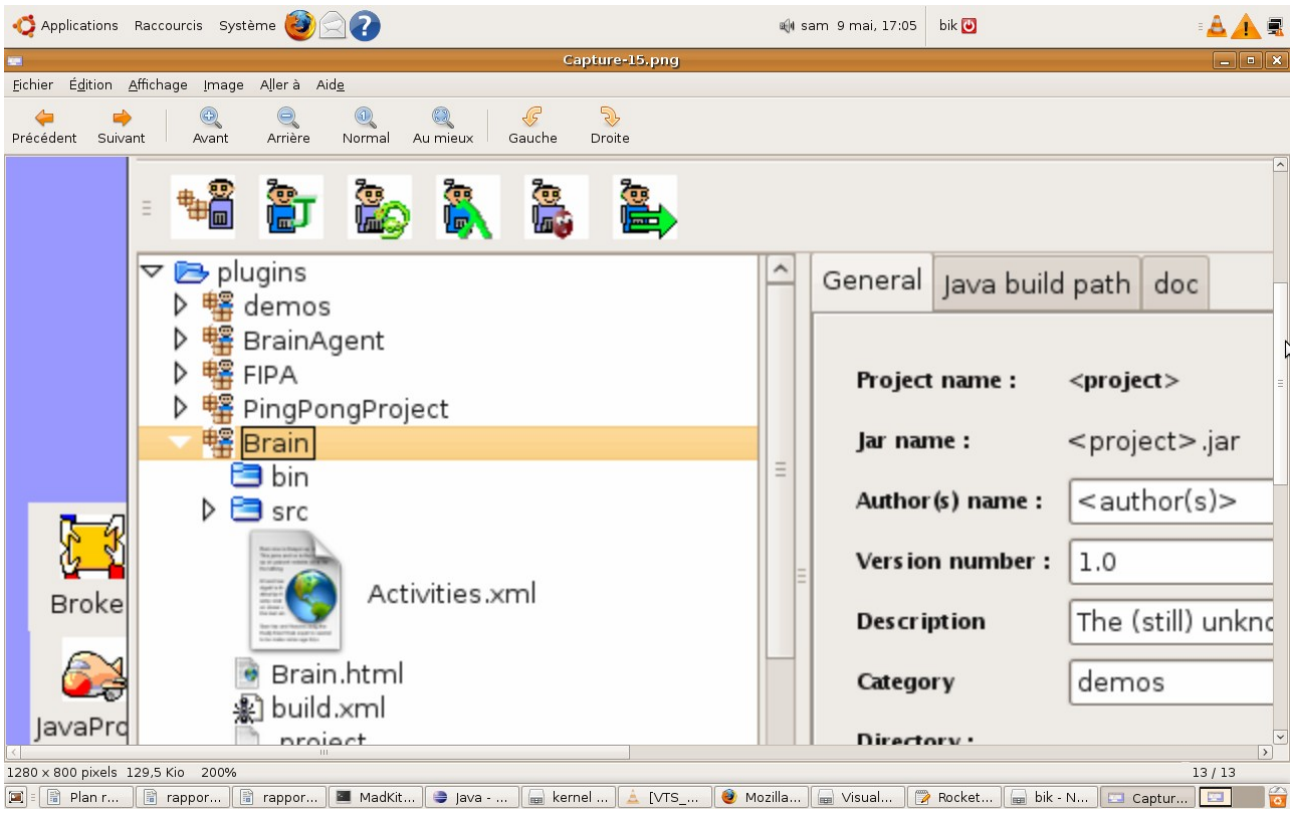
Voici au final la hiérarchie de classe à laquelle nous sommes parvenus:



III. Intégration du plugin dans Madkit

1. Création du projet Brain

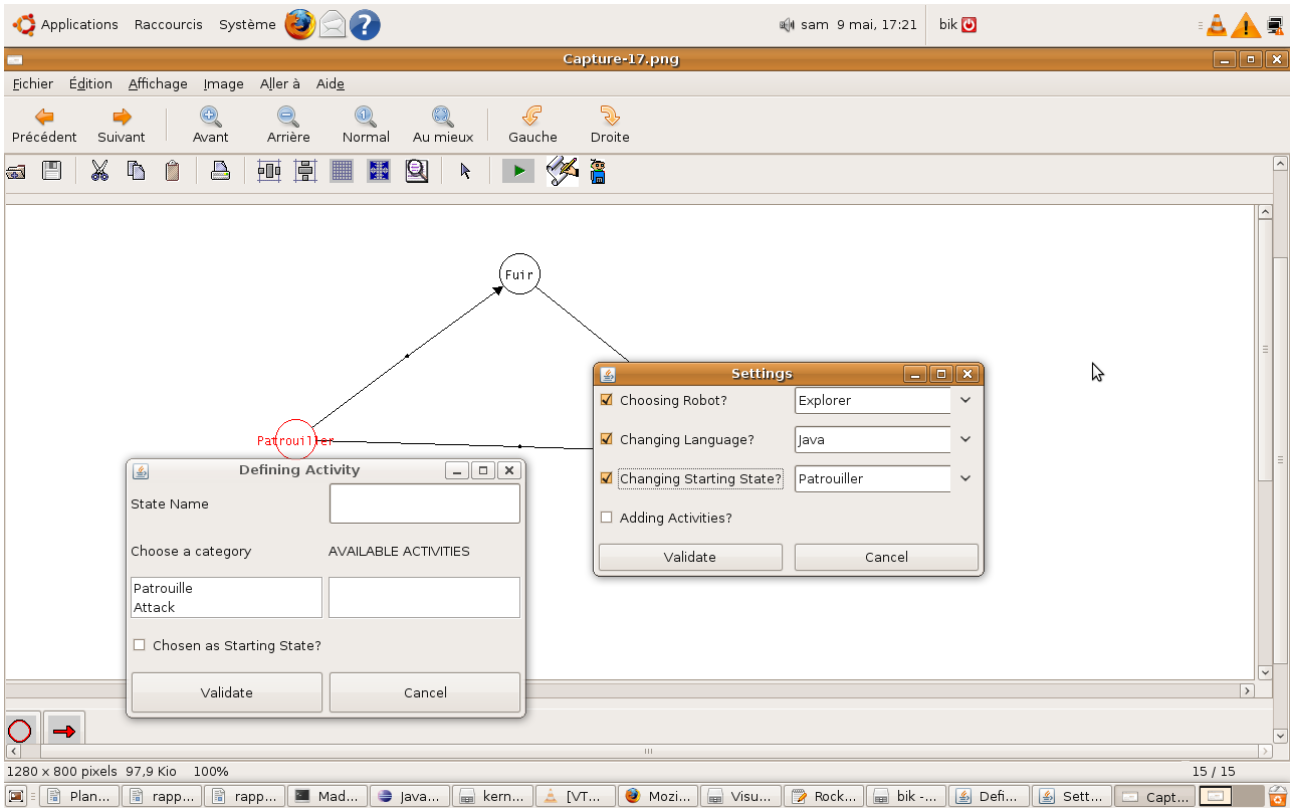
La première chose à faire c'est de créer un projet sous Madkit: Brain.



Créer un projet Java sous Eclipse et lui donner comme répertoire sources Brain qui se trouve MadKit/plugins/Brain
Ajouter les différents jar qui se trouvent dans le fichier build.xml du projet nouvellement créé sous Eclipse. On peut citer entre autre
Sedit.jar
jdom.jar
makitkernel.jar
madkitutils.jar
Nos fichiers XML sont à mettre dans MadKit/plugins/Brain
Le fichier formalisme Brain.fml est à mettre dans MadKit/lib/formalisms
Ainsi l'exécution se fait grâce à Explorer et par doubleClick sur Brain

2. Utilisation

Un doubleClick sur Brain affiche :



Pour finir, nous dirons que nous avons implémenté une interface faisant la même chose que SEdit. En effet, nous nous sommes servis de l'objet JGraph. Ayant beaucoup plus avancé sur SEdit qui est aussi le designer standard sous Madkit, nous avons jugé utile de finir avec. Voici l'interface de JGraph.

IV. Génération du fichier de Brain

Le but de notre projet est de générer du code source python initialement, mais nous avons aussi ajouté une génération de code Java. Le principe de la génération de code est simple, on part du fichier au format xml de la machine à états, on le parse afin de récupérer toutes les informations sur les états et les activités à exécuter pour chacun d'eux.

Nous avons prévus des classes dédiées à la génération de code en python comme en java, il est à noter que certaines opérations seront communes aux 2 langages. Après analyse du travail à réaliser, il s'est avéré que pour modéliser et implémenter les changements d'états d'un objet, il existait déjà des solutions dont celle du pattern State parfaitement adapté à nos besoins. Au final, nous générerons du code source utilisant l'architecture du pattern State. Une fois fixé sur la forme du code source généré, il sera plus simple de créer les classes de génération quelque soit le langage cible.

1. Utilisation pattern State

Le pattern State permet de changer le comportement d'un objet en fonction de l'état dans lequel il se trouve. Le changement d'état se fait après que des conditions de transitions se soient réalisées. L'adaptation de ce pattern est direct par rapport au travail à réaliser.

Le pattern comprends 3 classes de base qui sont nécessaire à son fonctionnement :

- la classe State est une classe abstraite de base modélisant tous les types d'états possibles, elle n'est jamais directement utilisée, elle permet juste de s'abstraire des différences d'implémentation entre états concrets.
- Les classes ConcreteState, ces classes représentent les états qui seront effectivement utilisées par l'objet les utilisant. Ils disposeront des activités à effectuer dans l'état courant.

- La classe Context, elle représente l'objet qui sera amenés à changer d'état, elle a une vision sur l'ensemble des états possibles qu'elle peut prendre.

Cette structure est une bonne base de travail, mais en aucun cas suffisante pour effectuer le changement de comportement du warbot. Nous allons donc le modifier et l'adapter à l'utilisation que nous voulons en faire.

2. Adaptation du pattern State

A. Classe Context

La classe Context sera dans notre cas la classe de Brain qui est utilisé dans Warbot pour définir les comportements des robots. Elle contiendra comme attribut principal un état de type State, ce sera l'état courant du robot et celui-ci effectuera l'activité correspondante. Cet état sera initialisé dans la fonction activate() du Brain. Les essais de changement d'état seront fait dans la méthode doIt(), c'est aussi dans cette fonction que l'activité sera effectivement exécutée.

B. Classe State

La classe State représente un état quelconque, c'est une classe abstraite, elle n'est donc jamais instanciée par contre elle définit la structure de base que tous les états doivent respectés. En effet, elle dispose des méthodes indispensables pour le passage d'un état à l'autre.

i) La méthode changeState()

C'est la méthode qui est chargée de savoir si le warbot doit changer d'état ou non. Pour ce faire, elle va déterminer les transitions possibles depuis l'état courant vers tous les états suivants possible indiqués dans le fichier xml de la machine à état. Il n'y qu'une façon de procéder, il faut tester les conditions de toutes les transitions, ensuite parmi les transitions retenues, on prends celle de plus haute priorité. Dans le cas où les transitions ont des priorités égales, on en prend une au hasard. Elle retourne une instance de l'état suivant et null si aucune transition possible.

ii) La méthode testConditions()

C'est la méthode utilisée par changeState() pour savoir quelle condition a été respecté. Elle prends le numéro de la transition en paramètre et retourne 1 si la condition est respecté, 0 sinon. Toutes les conditions de transition de cet état sont en opérande d'un if car elles sont récupérer telles que l'utilisateur les as entrées, on connait la correspondance numéro de transition condition car ils ont été coder en dure dans le fichier qui devra être généré.

iii) La méthode executeTransition()

C'est la méthode utilisée par changeState() pour retourner l'état suivant en fonction du numéro de transition.

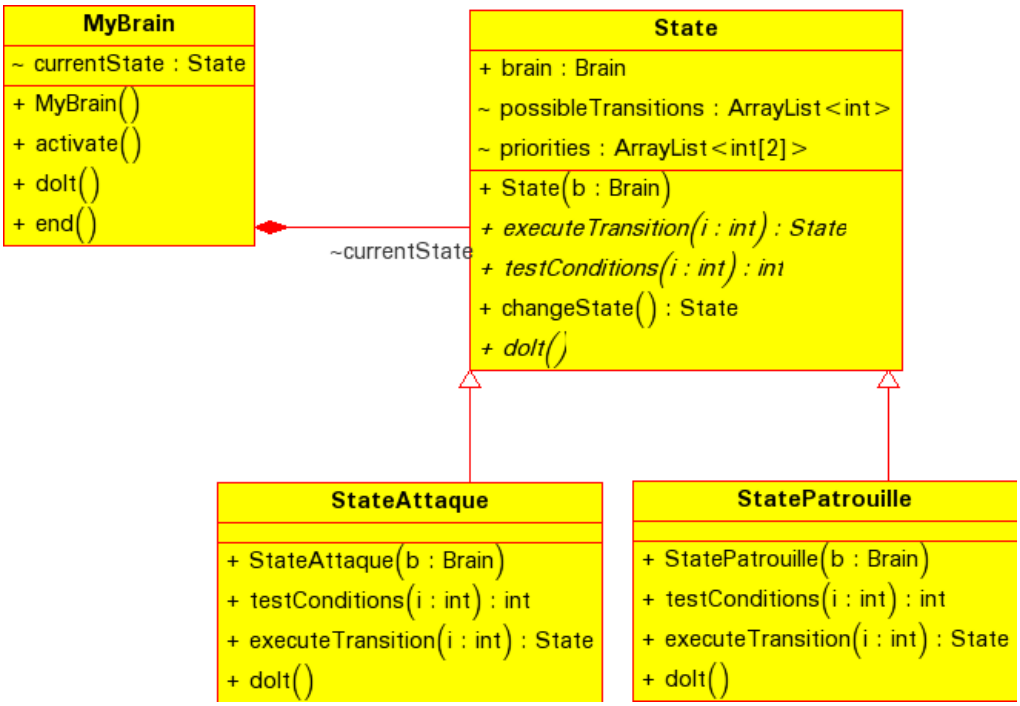
iiii) La méthode doIt()

C'est la méthode qui appelle l'activité que le warbot devra effectuer dans l'état courant.

C. Classes ConcreteStateX

Ou X est un langage. Ces classes sont les états concrets dans lequel l'agent pourra être. Elles héritent de la classe State pour disposer des méthodes de changement d'état et contiennent aussi le code des activités. Ce sont donc ces classes qui seront utilisées dans la pratique, chacune étant spécifique à chaque langage, elles définissent donc certaines caractéristiques du langage cible telles que l'indentation (indispensable dans le cas de python), les opérateurs et toute la syntaxe du langage en général.

Voici un exemple du pattern utiliser pour une machine à 2 états:



Nous avons effectué les modifications nécessaire au pattern State, ce qui nous donne une vision du code qui devra être généré dans le langage cible. Cette structure a les défauts de ces avantages, c'est à dire que le code généré sera lisible et facilement compréhensible pour l'utilisateur, ce qui facilitera sa modification éventuelle. L'inconvénient est que tout a été pensé pour un langage cible orienté objet, un portage de la solution que nous avons retenue ne sera donc utilisable que pour les langages de type orientés objets.

3. Génération du code

Maintenant que la structure du code source dans le langage cible est défini, il nous reste plus qu'a créer les classes de génération en java et python qui utiliseront du code implémentant notre pattern. Nous savons qu'il nous faudra générer le code dans 2 langages cibles que sont Java et Python, ils ont pour point commun d'être orientés objets, qualité indispensable pour être des langages cibles de génération.

Le langage générateur est dans les 2 cas le langage Java car c'est celui qui a été utilisé pour le développement de Madkit. Le principe de base de génération d'un langage à un autre est simple, tous les langages disposent de fonctions d'écritures vers une sortie que ce soit une console système ou un fichier des données de type texte, grâce à cela nous pouvons écrire dans un fichier du code écrit par un langage source en respectant la syntaxe d'un autre langage. Voici un exemple de la manière de procéder pour générer un petit code python à partir de Java :

```
PrintWriter fState = new
PrintWriter(newFileWriter("nomFichierPython.py"));
fState.print("print 'du code python'");
fState.close();
```

Dans cet exemple le fichier nomFichierPython est créé ou effacé si il existe déjà, on écrit ensuite le code python qui affichera « du code python » sur la console.

Nous avons maintenant une méthode de génération de code source. Il faut maintenant s'intéresser à ce qui va être généré à savoir la transformation du fichier xml de la machine à état en code python ou java utilisable dans Warbot. Pour le parcours du fichier xml, nous avons utilisé la bibliothèque Jdom qui permet la lecture et la récupération des données xml en les mettant dans un structure d'arbre qu'il nous suffira simplement de parcourir selon nos besoins. Il faut aussi tenir compte des fichiers contenant toutes les activités possible ainsi que le fichier d'indices pour l'intégration directe du code des activités; et ce, quelque soit le langage

cible. Ce facteur commun nous amène à envisager la factorisation du code java, ce qui à donné naissance à une hiérarchie de classe avec une classe générique qui se charge de la lecture des différents fichiers xml et pour chaque langage cible possible un classe fille qui génère effectivement le code dans le langage en question.

A. Classe XmlTo

C'est la classe abstraite qui a pour rôle d'ouvrir les 3 fichiers xml à savoir fsm.xml, le fichier de la machine à états, le fichier activities.xml et indices.xml. Elle récupère les données telles que les différents états de la machine dont celui de départ, ainsi que les transitions possibles. Elle récupère le code des activités en fonction du langage et de leur présence dans la machine courante.

Elle définit aussi l'indentation qui sera utilisée dans le langage cible de même que le type de warbot pour lequel le brain sera généré car certaines méthodes ne sont accessibles qu'à certain type de robots.

Ainsi, une fois sorti du constructeur de la classe, on sait ce qu'on a à générer, dans quel langage, à partir de quelles données et enfin dans quel fichier. XmlTo définit aussi des méthodes abstraites qui structure le code généré dans le langage cible pour être adapté aux méthodes basique des Brains de warbot. Dans le cas de Java, on doit hériter de la classe warbot.Brain pour pouvoir utiliser toutes les méthodes d'actions du corps.

Ce n'est pas nécessaire dans le cas de python car madkit a été prévu pour qu'on puisse appeler les méthodes sans héritage grâce à Jython qui un python totalement intégré à Java qui permet par exemple d'hériter de classe Java ou encore d'utiliser des objets java en python. On se contentera donc de faire un fichier python pure, le module jython de Madkit nous permettant d'utiliser l'api java indirectement.

Que le fichier généré soit en python ou en Java, 3 fonctions sont indispensables :

- doIt(), qui à chaque appel de l'ordonnanceur de la partie effectura les actions qui y sont programmées, par exemple errer au hasard sur la carte puis faire feu au premier ennemi en vue.

- activate(), qui est appelée une fois en début de partie pour chaque warbot, elle est chargée d'initialiser les différents paramètres ajouter par l'utilisateur.

- end(), qui est appelée lorsqu'un warbot est détruit.

Nous prendrons donc en compte ces contraintes lors de la génération de code et insérerons le pattern state de manière à ce qu'il fonctionne en accord avec ces trois méthodes. En outre, il a été aussi nécessaire de créer des méthodes supplémentaires pour nous aider dans notre travail, elles seront présentées dans la suite.

Une fois la classe XmlTo créée, on hérite d'elle pour créer une classe de génération dans le langage cible. XmlToJava et XmlToPython sont comme leur nom l'indique des classe filles de XmlTo qui auront a surcharger les méthodes abstraites de XmlTo. Ces méthodes comme nous l'avons dit plus haut sont abstraites et doivent être surchargé pour qu'elle génère du code respectant une syntaxe de langage. Nous allons présenter uniquement le travail fait en Python car il est pratiquement le même en Java et obéit aux même principes.

B. GenerateAbstractState()

Dans cette méthode la classe State du pattern exposé plus haut est générée. Il est à noter qu'elle a aussi la responsabilité de générer la méthode changeState() qui identifie les transitions possibles et retourne celle de plus haute priorité.

C. GenerateConcreteState()

Elle génèrent les états concrets qui sont décrient dans le fichier xml de la machine à états. Elle construit aussi 2 tableaux, un qui stockera les transitions possibles en fonction de leur numéro et le deuxième stockera les priorités. Le tableau des transitions sera remplit dynamique lorsque le code cible sera exécuté tandis que le tableau des priorités est initialisé manuellement, voici une illustration : self.priorities = ['10','10']. Enfin elle génére pour chaque état les méthodes executeTransition(), testConditions et doIt(). Leur fonctionnement général a déjà été expliqué, mais il est à noter qu'elle fonctionne en prenant toujours en paramètre un numéro de transition, nous sommes obligé de les générer avec une succession de « If » pour pouvoir faire l'association entre le numéro, la condition et la transition en question.

Ce code illustre notre propos :

```
def testConditions(self,i):
        if (i == 0):
            if ( len(self.brain.getPercepts()) > 2 ):
                return 1
            else:
                return 0
        if (i == 1):
            if ((len(self.brain.getPercepts()) <= 2 and
len(self.brain.getPercepts()) > 0)):
                return 1
            else:
                return 0
```

```
def executeTransition(self,i):
    if (i == 0):
        return StateFuite(self.brain)
    if (i == 1):
        return StateAttaque(self.brain)
```

On sait après l'appel de testConditions() quelle transition est possible car pour un i passé en paramètre, on retourne 1 si la condition est vérifié, donc la transition numéro i est éventuellement celle qui doit être exécutée. Éventuellement, car il se peut qu'une autre transition potentielle ait une priorité plus importante, et c'est le rôle de changeState() de déterminer cela.

GenerateConcreteState, écrit aussi le code de l'activité qui aura été préalablement récupérer dans le constructeur de la super classe et enfin elle écrit la méthode doIt() qui ne fait qu'appeler la méthode de l'activité.

Ces 2 méthodes sont le cœur de la génération du code de la machine à états finis, on peut gérer le passage d'un état à un autre et les activités qui doivent être faites dans ces états. Pour finir, il suffira de créer un nouvel attribut à la classe Brain qui sera l'état courant, cela est fait par la méthode generateCustomAttributs(), generateActivateMethod() initialise cet attribut à l'état de départ, generateDoItMethod() ne fait que tester si un changement d'état doit être effectuer et enfin generateEndMethod() ne fait qu'afficher un message de mort.

D. GenerateInfixedFormConditions()

Cette méthode récursive permet de passer de la forme d'arbre d'une expression logique qu'on a récupéré dans la balise <x-condition> à une expression en forme infixe. Il s'agit de faire un parcours d'arbre infixe en récupérant le type de condition et leur placement dans l'expression. Pour savoir qu'elle est le symbole dans le langage cible on utilise la méthode translateLogicalOperatorSymbol() qui doit être surcharger par les classes filles. Voici un exemple : <and-condition> sera traduit par « and » en python.

CONCLUSION

Nous avons pu par l'intermédiaire de Sedit fournir à l'utilisateur un moyen de définir graphiquement le comportement d'un warbot et ce sans avoir de notions en programmation. Cette représentation graphique est ensuite traduite en Java et Python conformément au cahier des charges.

Les fonctionnalités requise ont été implémentés mais il reste que nous aurions pu encore améliorer la qualité de notre travail avec un peu plus de temps. En effet, nous avions dans l'idée de permettre à un utilisateur avancé dans l'utilisation de madkit de pouvoir rajouter ses propres activités. C'est actuellement possible en modifiant les fichiers xml que nous avons définit, mais loin d'être pratique, l'idéal étant de fournir dans l'interface graphique un moment d'ajout direct d'une activité.

Au niveau du code généré, le code n'a pas pu être optimiser pour que à chaque changement d'états, l'agent n'ait pas a créer une nouvelle instance d'une classe d'état, la création d'instance étant couteuse en ressource système. Ce n'est pas un problème sur les ordinateurs actuels, mais un utilisateur avec un ordinateur vieillissant pourrait voir ce dernier victime de ralentissement avec un nombre important de warbot.

Pour ce qui est de l'interface, nous étions partis sur plusieurs pistes, nous avions envisagé notamment de nous passer de Sedit et de créer notre propre logiciel graphique. Un prototype en java swing utilisant JGraph a été réalisé mais il n'a pas été retenu car cela dépasser le cadre du projet, de plus nous n'avions pas trouvé de moyen facile de directement l'intégrer sous Madkit.

Nous avons apprécié travailler sur ce projet car il nous a permit de nous aguerrir dans la programmation orienté agent avec l'utilisation de la machine à états comme point de départ pour la modélisation de comportement et la création des activités pour les robots.

Enfin, il nous a fait découvrir les principes de la génération de code et par la même, apprendre le langage python avec lequel nous n'étions pas familiers.

Il en va de même pour la manipulation de fichier xml avec la bibliothèque Jdom qui nous a été très utile pour parser le fichier de la machine à états.

BIBLIOGRAPHIE

www.google.com
www.madkit.net
www.developpez.com
www.python.org
www.wikipedia.com
API Java 1.5
www.lirmm.fr/~ferber
www.warbot.fr