Programmation situationnelle : programmation visuelle de comportements agents pour non informaticiens

F. Michel[a] fmichel@lirmm.fr
J. Ferber[a] ferber@lirmm.fr
P.-A. Laur[b] pal@feerik.com
F. Aleman[b] florian@feerik.com

[a]LIRMM Laboratoire d'Informatique, Robotique et Micro-électronique de Montpellier
CNRS - Université Montpellier II, 161 rue Ada 34392 Montpellier Cedex 5 - France

[b]FEERIK, Inc. 91 rue Font Caude 34080 Montpellier - France

Résumé

Cet article présente une approche pour la programmation visuelle de comportement agent. L'objectif de cette approche appelée programmation situationnelle (PS) est de permettre à des utilisateurs non informaticiens d'élaborer facilement le comportement d'entités artificielles dans le contexte d'un domaine d'application particulier. Ainsi, la PS définit un ensemble de principes de conception permettant de développer des outils de programmation visuelle susceptibles d'être utilisés par des personnes n'ayant aucune connaissance de la programmation, ni du paradigme multi-agent. Dans cet article, nous présentons la PS et la manière dont elle a été utilisée pour développer un jeu vidéo en ligne basé sur la programmation visuelle de comportements agents, et qui peut être joué par un utilisateur lambda.

Mots-clés : *Programmation orientée agent, Programmation visuelle, jeu vidéo, simulation multi-agent*

Abstract

This paper presents an agent-oriented visual programming approach which aims at providing MAS end-users with a means to easily elaborate artificial autonomous behaviors according to a targeted domain, namely situational programming (SP). More specifically, SP defines design principles which could be used to develop MAS visual programming toolkits suited for non developers and MAS novices. This paper presents SP and how it is used to build a video game which can be played by MAS novices, that is any Internet user.

Keywords: *Agent-Oriented Programming, Visual Programming, Video Game, multi-agent based simulation*

1 Introduction

Le paradigme multi-agent est aujourd'hui utilisé dans de nombreux domaines de recherche et d'application. Notamment, la modélisation et la simulation de systèmes multi-agents (SMA) sont très utilisées pour étudier des phénomènes complexes dans des domaines tels que les sciences sociales, l'éthologie, l'économie ou encore la robotique mobile collective [12]. De cet aspect interdisciplinaire découle un problème récurrent quant à l'utilisation des SMA : alors que la plupart des outils existants nécessitent des connaissances en programmation, de très nombreux utilisateurs finaux ne sont pas des programmeurs expérimentés. De fait, alors que le potentiel du paradigme multi-agent est relativement simple à exposer et à partager, il est aujourd'hui difficile pour un novice en informatique de l'expérimenter concrètement, par lui-même, sur ordinateur. Dans la perspective d'une diffusion plus large des SMA dans la société, nous pensons que permettre à un utilisateur lambda d'expérimenter les principes de conception qui leur sont associés constitue un enjeu sociétal majeur (éducation, jeux vidéos, jeux sérieux, etc.).

Comme nous l'avons dit, la problématique de l'utilisation des SMA par des débutants en informatique est tout particulièrement criant dans le cadre de la simulation de SMA. Ainsi, conscients que leurs utilisateurs finaux ne sont pour la plupart pas informaticiens de formation, les développeurs de simulateurs multi-agents abordent ce problème en définissant un ensemble de primitives de codage qui facilite la programmation de comportements étant donné un domaine applicatif (e.g. Cormas [2]). Cependant, même si cela permet de cacher une partie de la complexité des langages de haut niveau qui sont effectivement utilisés, l'utilisateur doit connaître (1) les base de la programmation (expressions conditionnelles, boucles, uti-

lisation de variables, etc.) et (2) la syntaxe associée au langage défini par la plate-forme. C'est pourquoi de nombreux travaux se sont tournés vers des solutions intégrant de la programmation visuelle (PV). En effet, même si elle n'est naturellement pas aussi expressive que la programmation textuelle, la PV est une solution qui permet l'utilisation d'une plate-forme par des novices en informatique.

Ce papier présente une approche, la *programmation situationnelle* (PS), qui permet de définir des éléments de PV pour l'élaboration de comportements artificiels. Plus précisément, le but de la PS n'est pas de permettre le développement de zéro d'un SMA par un novice. La PS définit plutôt des principes de conception qui permettent de construire des outils multi-agents permettant aux utilisateurs n'ayant aucune connaissance de la programmation, ni des SMA, de programmer visuellement le comportement d'entités artificielles.

La section suivante présente l'intérêt des approches basées sur de la PV pour les SMA, quelques outils existants, puis discute de leurs limites. La section 3 présente les motivations et les idées sur lesquelles se base la PS. La section 4 détaille un cas d'étude réalisé dans le contexte d'un projet de jeu vidéo en ligne appelé Warbot. Les sections 5 et 6 discutent respectivement des limitations actuelles de l'approche proposée et des travaux de recherche qui peuvent lui être associés. Enfin, la section 7 conclue ce papier et donne quelques perspectives de recherche relatives à la PS.

2 Programmation visuelle et SMA

2.1 Motivations

La PV permet aux utilisateurs finaux de créer des comportements d'agents artificiels grâce à des éléments graphiques correspondant à des blocs de programmation textuelle. Dans la plupart des cas, ces éléments graphiques sont connectés entre eux par des flèches représentant des relations telles que l'ordonnancement des actions, une expression conditionnelle, des boucles, etc. Ainsi, les utilisateurs peuvent construire un diagramme représentant le comportement global d'un agent sans avoir à connaître le langage de programmation effectivement utilisé par la plate-forme.

Un autre avantage majeur de la PV repose sur le fait qu'elle permet d'abstraire l'utilisateur des problèmes de syntaxe liés à l'utilisation d'un langage de programmation. Cela grâce au fait que les éléments graphiques manipulés ne représentent que des instructions valides. De plus, les outils de PV reposent généralement sur une grammaire qui permet de contrôler la validité du contenu des éléments graphiques et des connexions qui les relient, ce qui constitue une aide supplémentaire pour l'utilisateur.

Les principaux avantages des outils de PV peuvent donc être résumés ainsi :

1. Connaître le langage de programmation utilisé dans la plate-forme n'est pas nécessaire.
2. La validité de la syntaxe peut être assurée grâce à la grammaire sous-jacente à la manière dont les éléments graphiques peuvent être définis et combinés.

2.2 Exemples d'outils de programmation multi-agent visuelle

Cette section présente trois exemples de plates-formes multi-agents possédant des outils basés sur de la PV : (1) AgentSheets, (2) SeSAm et (3) Repast Simphony.

AgentSheets. Développé au début des années 90, l'idée fondatrice du logiciel de programmation multi-agent AgentSheets est de fournir, à des utilisateurs non professionnels, un média informatique leur permettant de construire des simulations multi-agents interactives [14]. Ainsi, la philosophie de cet environnement consiste à cacher le plus possible la complexité des mécanismes liés à l'élaboration d'une simulation multi-agent. Cela afin de promouvoir l'utilisation de la simulation multi-agent comme un outil de réflexion qui puisse être utilisé dans le plus de domaines possibles. AgentSheets est notamment très utilisé dans un cadre pédagogique.

Le langage de PV d'AgentSheets est appelé Visual AgenTalk (VAT). VAT est un langage à base de règles qui permet aux utilisateurs d'exprimer des comportements agent sous la forme de règles *si-alors* liant des conditions à des actions. De plus, VAT permet d'augmenter l'interactivité de la PV grâce à ce qui est appelé le *Tactile Programming* : (1) des fragments de programmes peuvent être manipulés grâce à des opérations de glisser/déposer pour composer des comportements et (2) la programmation est guidée grâce à de nombreuses opérations disponibles avec

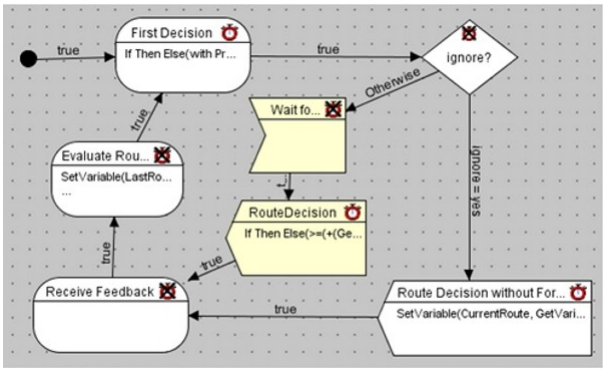

FIGURE 1 – Copie d'écran du comportement d'un agent conducteur dans SeSAm [10]

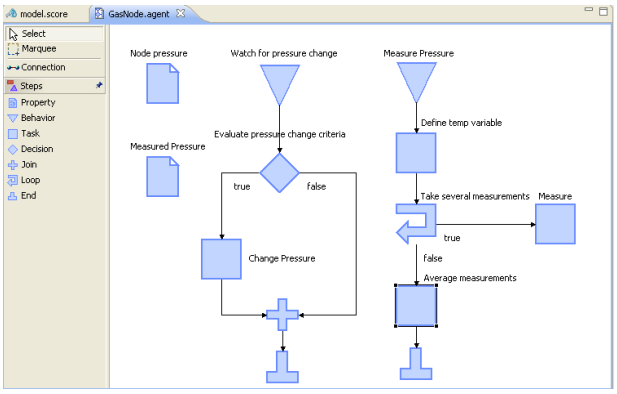

FIGURE 2 – L'éditeur graphique de comportement agent de la plate-forme Repast Simphony

de simples clics de souris. AgentSheets [1] est un projet commercial qui existe encore aujourd'hui et qui a récemment été décliné dans une version 3D appelée AgentCubes [8].

SeSAm. (Shell for Simulated Multi-Agent Systems) est un logiciel open source qui fournit un environnement de simulation multi-agent générique [9]. SeSAm [2] contient plusieurs outils de PV qui facilite la modélisation de comportements agents.

Par exemple, des boîtes de dialogues interactives permettent de spécifier des primitives de codage, potentiellement récursives, qui peuvent être utilisées pour définir des règles comportementales, des procédures d'initialisation, etc. Ces boîtes de dialogues cachent non seulement à l'utilisateur l'utilisation du langage Java mais vérifient aussi la validité des primitives qui sont crées. Notamment, la consistance entre les types de données est vérifiée. Ainsi, des comportements agents peuvent être programmés sans rien connaître des subtilités d'un langage de programmation orienté objet. Dans SeSAm, la PV est aussi utilisée pour visualiser et spécifier le comportement global d'un agent grâce à des diagrammes d'activité comme le montre la figure 1 : les nœuds représentent des activités et sont reliés par des flèches symbolisant des règles de transition entre ces activités.

Repast Simphony. (RS) est un logiciel open source dédié à la modélisation et à la simulation de SMA. RS, extension de la plate-forme Repast [3] à destination des utilisateurs non informaticiens, fournit de nombreux outils facilitant la modélisation et la programmation visuelle de SMA. Techniquement, RS est une version préconfigurée de l'IDE (Integrated Development Environment) Eclipse qui utilise notamment un greffon appelé Flow4J [4] qui permet de modéliser visuellement des processus grâce à la souris.

Comme le montre la figure 2, Repast fournit lui aussi un éditeur permettant de définir la logique comportementale d'un agent grâce à des diagrammes composés de blocs représentant différents traitements : perception, action, expression conditionnelle, boucle, etc.

Ainsi, l'utilisateur n'a pas besoin d'écrire une seule ligne de code : le diagramme est automatiquement convertit dans un langage de programmation traditionnel.

2.3 Limites des approches existantes

Complexité comportementale. Comme le souligne [15], la PV de modèles cognitifs complexes reste une tâche extrêmement difficile. En effet, les outils de PV existants ne manipulent le plus souvent que des concepts très basiques. Pour pallier ce problème dans le contexte des sciences sociales, [15] propose un langage de modélisation visuel qui repose sur la méthodologie INGENIAS [13]. Cette approche permet notamment de définir graphiquement des comportements complexes basés sur la notion d'intention et l'utilisation de concepts organisationnels. Cependant, comme le soulignent ses auteurs, ce langage reste sophistiqué dans la mesure où il vise intentionnellement les experts en modélisation multi-agent, notamment ceux issus des sciences sociales. Néanmoins, dans le cadre de la PV de comportements agents, ce

1. http ://www.agentsheets.com
2. http ://www.simsesam.de
3. http ://repast.sourceforge.net
4. http ://flow4jeclipse.sourceforge.net

travail montre clairement les limites des outils existants en terme d'expressivité.

Représentation graphique du comportement. La difficulté de programmer visuellement des comportements complexes ne tient pas seulement à la relative simplicité des concepts généralement manipulés. En effet, représenter graphiquement un comportement à l'écran est une tâche délicate. Notamment, outre la taille nécessaire à sa représentation, un diagramme composé de nombreux éléments graphiques ne peut pas vraiment être explicite, ni intuitif, et donc compréhensible pour un non initié. De plus, même s'il est possible de (1) réduire les éléments graphiques à des icônes ou (2) d'utiliser des briques de programmation définies récursivement, la taille de l'écran reste une limitation incontournable. C'est pourquoi il est fondamental de fournir aux utilisateurs finaux des abstractions graphiques permettant de représenter de manière simple et synthétique la logique comportementale d'un agent.

Connaître la programmation reste nécessaire. Au delà des limites citées jusqu'à présent, nous pensons que le problème majeur des approches existantes reste le suivant. Même si un très haut niveau d'abstraction est considéré, les outils de PV existants nécessitent la compréhension des concepts de base de la programmation. Malgré leur aspect intuitif, ces outils font en effet toujours appel à des concepts tels que les expressions conditionnelles du type *si-alors-sinon*, les boucles, la déclaration et l'assignation de variables, etc.

Dans un contexte pédagogique, cela peut ne pas être un problème dans la mesure où l'objectif peut précisément être l'apprentissage de ces concepts, voire de la programmation multi-agent elle-même. Cependant, dans la perspective de l'appréhension des SMA par un utilisateur lambda, l'utilisation de ces concepts constitue bien un verrou majeur : l'utilisateur est obligé de les connaître et de bien les comprendre.

C'est pourquoi nous pensons que l'élaboration d'outils de PV pour SMA ne faisant pas appel à de tels concepts constitue un enjeu majeur pour la promotion des SMA. La section suivante présente une approche de PV pour comportements agents qui ne nécessite aucune connaissance de la programmation, ni des principes de modélisation multi-agent : la *programmation situationnelle* (PS).

3 Programmation situationnelle

3.1 Objectif

L'objectif principal de la PS est de permettre à des non informaticiens, n'ayant pas de connaissance du paradigme multi-agent, de facilement programmer et tester des comportements artificiels dans le cadre d'un domaine ciblé. Manifestement, un tel but requiert tout d'abord une approche basée sur la PV. De plus, cette approche doit répondre autant que possible aux problèmes que nous avons soulevés dans la section précédente. Les objectifs de la PS peuvent ainsi être résumés en trois points :

- Les utilisateurs ne doivent être confrontés à aucun concept de programmation traditionnelle, ni à des tâches de modélisation complexes.
- La représentation de la logique comportementale de l'agent doit prendre peu d'espace à l'écran et donc être la plus synthétique possible.
- La définition de comportements complexes doit être possible.

Ainsi, contrairement à certaines des approches décrites précédemment, notre but n'est pas de fournir aux non développeurs des moyens de réaliser un SMA ex nihilo. Le propos de la PS est plutôt de définir des principes de programmation orientée agent qui peuvent être utilisés pour développer des outils de PV qui remplissent effectivement les objectifs que nous avons énoncés.

3.2 Principe de la PS

Comme nous allons maintenant l'expliquer, l'idée générale de la PS n'est pas uniquement liée aux problématiques sous-jacentes à la PV. Celle-ci repose sur une réflexion plus générale à propos de la programmation orientée agent.

Considérons donc tout d'abord un modèle traditionnel et synthétique du processus comportemental d'un agent comme celui illustré par la figure 3. Celui-ci peut être considéré comme composé de trois phases distinctes : (1) perception, (2) délibération, puis (3) action [4].

D'un point de vue technique, programmer le comportement d'un agent repose donc sur (1) le traitement des perceptions, (2) l'utilisation de ces résultats dans la délibération et (3) la concrétisation de cette dernière par une action de l'agent sur son environnement.

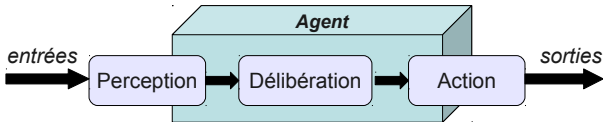

FIGURE 3 – Le comportement d'un agent : (1) Perception (2) Délibération (3) Action

Ici, il faut tout d'abord remarquer que, à partir d'un ensemble de percepts bruts, il peut être difficile de construire programmatiquement une vue pertinente de l'état du monde. Autrement dit, il est dans un premier temps déjà nécessaire de travailler sur les percepts afin de construire des informations pertinentes pour le processus de délibération. Cette tâche peut rapidement devenir ardue en fonction du niveau de complexité du comportement de l'agent. Par exemple, pour un robot footballeur, il est très difficile de capturer les aspects importants d'une situation à partir de ses données sensorielles [1].

Le même type d'observation est aussi valable à propos des actions et des plans qui doivent être définis pour un agent : ils peuvent être très complexes à programmer à partir d'un ensemble d'actions de base manipulant directement les effecteurs d'un agent.

C'est pourquoi, afin de remplir ses objectifs, la PS repose sur une approche où la perception et l'action sont considérées comme des traitements trop complexes pour un utilisateur lambda. De plus il faut remarquer que ces deux phases ne constituent pas les parties discriminantes pour un type d'agent particulier : modulo une paramétrisation, l'ensemble des percepts et des actions possibles reste le même pour deux agents ayant les mêmes capacités, seule la manière dont ils délibèrent les différencie vraiment. L'idée de la PS est donc de pré-programmer les phases de perception et d'action afin que l'utilisateur puisse se concentrer sur la partie délibérative de l'agent.

3.3 Perceptions et actions de haut niveau

La complexité liée à l'utilisation de percepts de bas niveau par le processus délibératif d'un agent est par exemple discuté de manière très intéressante dans [3]. Ce travail propose une architecture cognitive composée de trois niveaux : (1) *Reality*, (2) *Concept* et (3) *Mind*. Le niveau *Concept* est d'un intérêt tout particulier car il est chargé de faire la correspondance entre la réalité du monde physique et des concepts de haut niveau qui seront utilisés par le niveau *Mind*. En

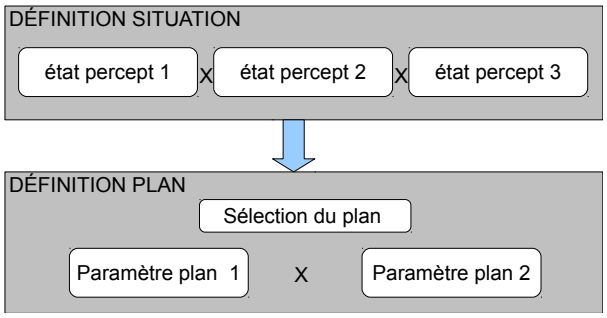

FIGURE 4 – Principe général de la programmation situationnelle

d'autres termes, le but du niveau *Concept* est de faciliter le travail de délibération du *Mind* en lui fournissant des concepts particulièrement adaptés à ce processus. L'idée majeure de la PS repose sur une approche similaire.

L'idée est de fournir aux utilisateurs des percepts d'un haut niveau d'abstraction lui permettant de définir des situations calibrées pour un domaine particulier. Par *situation*, nous signifions une combinaison des états possibles de chaque percept à partir de laquelle l'utilisateur n'aura plus qu'à choisir une action correspondante. Par exemple, un percept peut être *être attaqué*, ou encore *posséder le ballon* et son état dans la situation *vrai, faux*, ou *ignoré*. Ainsi, la définition par l'utilisateur d'une situation revient simplement à la sélection d'un état pour chaque percept.

Le même principe est utilisé pour décrire les actions qui sont associées par l'utilisateur aux situations qu'il définit. En effet, au lieu d'actions de bas niveau, l'utilisateur sélectionne des plans d'action de haut niveau spécifiés comme des combinaisons d'actions prédéfinies facilement paramétrables. Par exemple, un plan peut être *patrouiller une zone* (plan principal) en effectuant un *mouvement sinusoïdal* (un des paramètres du plan).

Par ailleurs, bien que l'utilisateur puisse définir autant de situations qu'il le souhaite, l'intérêt est qu'il ne se concentre que sur une situation à la fois, ce qui facilite la compréhension de la programmation du comportement. Nous discutons plus loin de la manière dont les éventuels conflits entre situations sont résolus par l'utilisateur. La figure 4 donne une représentation schématique de cette approche.

Le travail de programmation réalisé par l'utilisateur est donc simple et intuitif : (1) définir une situation en paramétrant les différents percepts,

puis (2) décider du plan associé et de ses paramètres. Ainsi, en définissant plusieurs couples situation/plan, l'utilisateur définit, et donc programme, le comportement d'un agent en suivant simplement son propre mode de pensée, matérialisant ainsi un processus de délibération très complexe, le sien, sans avoir à manipuler de notions algorithmiques pour autant.

3.4 Complexité des comportements sans complexité de la GUI

A première vue, le nombre possible de couples situation/plan représente le seul facteur de complexité pour un comportement. La PS peut donc sembler limitée en termes de possibilités étant donné que les paramètres des situations et des plans sont prédéfinis. Cependant, nous allons présenter d'autres principes de conception, inclus dans la PS, qui permettent d'élaborer de nombreux comportements différents sans complexifier l'interface graphique utilisateur (GUI).

Définition d'une situation. Comme expliqué précédemment, l'état de chaque percept contribue à la définition d'une situation. Les percepts les plus simples sont liés à la véracité d'une proposition et possèdent uniquement trois états : (1) vrai, (2) faux ou (3) ignoré (e.g. *être attaqué*). Pour de tels percepts, la sélection de leur état est réalisée à l'aide de simple clics faisant permuter le percept d'un état à un autre. Dans la PS, ce type de percepts est qualifié de *booléen*.

Par contre, en ce qui concerne des percepts liés à une donnée quantitative telle que le niveau d'énergie, l'état du percept (e.g. *énergie basse* ou *énergie haute*) doit être défini selon un seuil (e.g. 50%). Un tel seuil peut être défini en interne et volontairement caché à l'utilisateur. Mais lorsque celui-ci est présenté explicitement, l'idée est de donné à l'utilisateur l'opportunité de choisir manuellement le seuil grâce à un simple curseur graphique. Dans la PS, ce type de percepts est qualifié de *seuillé*. Ainsi, lorsqu'une situation contient un percept seuillé, le nombre de situations qui peuvent être définies est potentiellement infini. Cela permet donc d'introduire un facteur de complexité pour la définition des comportements sans que la simplicité de la GUI n'ait à en souffrir. Point très important, cela induit aussi de l'hétérogénéité et de la singularité entre les comportements qui peuvent être programmés par les utilisateurs.

Paramétrisation d'un plan. Les plans sont définis à un très haut niveau d'abstraction, en fonction du domaine considéré. Par exemple, le plan *patrouiller une zone* peut avoir deux paramètres : (1) la localisation de la zone et (2) le type de mouvement utilisé (sinusoïdal, ligne droite, etc.). Beaucoup de complexité peut être obtenue en fonction de la latitude que donne les paramètres d'un plan. Cependant, cet aspect est entièrement dépendant du domaine visé et il ne peut être généralisé de la manière dont nous l'avons fait pour les percepts. Malgré tout, nous avons identifié un paramètre générique très intéressant qui peut être utilisé quel que soit le domaine considéré.

En effet, un des problèmes que nous avons rencontrés au début de ce projet était que, dans certains cas, les agents changeaient constamment de plan : une situation pouvant disparaître puis réapparaître en très peu de temps. Pour solutionner ce problème, nous avons alors ajouté à tous les plans un paramètre générique : *l'obstination*. L'obstination définit combien de temps un agent doit conserver un plan avant de reconsidérer la situation courante. L'obstination est réglée, pour chaque plan, par l'utilisateur grâce à un simple curseur graphique. Ce paramètre permet ainsi de résoudre, dans le cadre de la PS, un problème bien connu en ce qui concerne la sélection d'actions, notamment identifié par Tyrrell dans [17] comme le problème de la persistance et de l'engagement dans la sélection d'action (*persistence/commitment in action selection*).

De plus, L'obstination permet d'introduire un nouveau facteur de complexité qui augmente lui aussi la singularité des comportements produits par l'utilisateur, donnant ainsi en quelque sorte aux agents de la personnalité.

Priorités entre couples situation/plan. Un aspect important de la PS dont nous n'avons pas encore parlé est celui des conflits entre situations. L'utilisateur peut en effet définir des situations différentes qui peuvent correspondre simultanément à un état du monde particulier. C'est pourquoi, il est important de fournir à l'utilisateur un moyen d'introduire des priorités entre situations grâce à une représentation graphique simple. Pour ce faire, une colonne dans laquelle les situations (résumées) sont ordonnées est apparue comme une solution bien appropriée. Une telle présentation permet à l'utilisateur de définir les priorités entre situations grâce à des actions du type glisser/déposer. De plus, cela permet encore une fois d'augmenter le nombre de comportements qui peuvent être définis pour un coup très modeste en terme de complexité de la GUI.

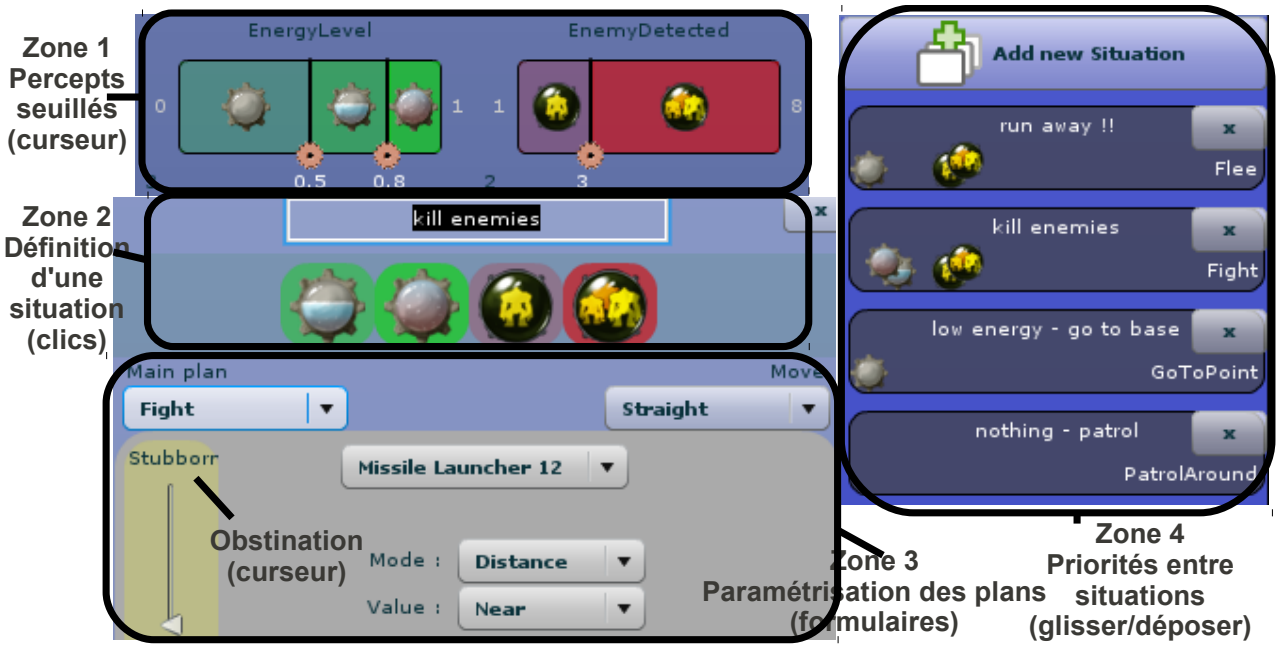

FIGURE 5 – Programmation situationnelle avec l'interface graphique de Warbot

4 Cas d'étude : le jeu vidéo Warbot

4.1 Historique et objectifs

Warbot [5] est un jeu vidéo dans lequel deux équipes de robots combattent l'une contre l'autre. Historiquement créé il y a une dizaine d'années à l'aide de la plate-forme MadKit [5], Warbot est encore utilisé aujourd'hui pour apprendre aux étudiants en informatique la programmation multi-agent à travers une compétition. L'idée est que les étudiants programment uniquement le *cerveau* des agents, c'est-à-dire leurs comportements, tandis que leur *corps* est prédéfini, de telle sorte que les meilleurs comportements gagnent la bataille.

En collaboration avec la société Feerik [6], spécialisée dans le jeu vidéo en ligne, nous développons une version de Warbot qui puisse être jouée par tout utilisateur, et en particulier les non informaticiens : un joueur Feerik peut être n'importe quel utilisateur d'Internet.

4.2 Percepts et plans dans Warbot

Comme discuté dans la section 3, développer un outil informatique basé sur la PS suppose d'abord d'identifier les percepts et les plans de haut niveau associés au domaine ciblé. À partir de l'expérience que nous avons de la version pédagogique de Warbot, nous avons identifié un premier ensemble de cinq percepts régulièrement programmés par les étudiants tout au long de ces dernières années : *Niveau d'énergie*, *nombre d'ennemis détectés*, *être attaqué*, *être aidé par un coéquipier* et *un coéquipier appelle à l'aide*.

De façon similaire, quatre plans de haut niveau ont été identifiés : *Rejoindre une zone*, *Patrouiller*, *Combattre* et *Fuir*. Ici, on peut se demander pourquoi il n'y a pas de plan *aider un coéquipier*. En fait, il a été décidé que cela constituerait un paramètre commun à tous les plans. Autrement dit, lorsqu'il applique un plan, un robot peut décider de prendre en compte ou non les desiderata de ses coéquipiers.

La figure 5 présente des portions de l'éditeur de comportement de la nouvelle version de Warbot. La zone 1 présente les percepts que l'utilisateur peut sélectionner pour définir une situation dans la zone 2 (glisser/déposer). La zone 2 permet aussi de nommer la situation courante. La zone 3 permet de paramétrer le plan associé à la situation. La zone 4 résume les situations déjà définies et permet de les ordonner (glisser/déposer) : la plus prioritaire en haut. D'autres pages web sont utilisées pour définir les robots qui composent l'équipe, ainsi que les capacités de leur équipement (jambes, bras, armes, etc.).

5. www.madkit.net/warbot
6. www.feerik.com

FIGURE 6 – Rendu 3D du jeu Warbot à l'aide du lecteur web Unity

La figure 5 montre aussi que, lorsqu'il définit des situations, l'utilisateur peut sélectionner plusieurs états du même percept afin d'associer facilement différentes situations à un unique plan en une seule fois : dans cet exemple le robot combattra quel que soit le nombre d'ennemis s'il a plus de 50% d'énergie.

L'interface graphique actuelle n'inclut volontairement pas tous les percepts et plans que nous avons précédemment mentionnés. En effet, il est prévu d'introduire de manière incrémentale chaque percept, selon un scénario ludique, afin de familiariser progressivement l'utilisateur avec la programmation d'un robot. La simplicité, au premier abord, du jeu est en effet une priorité pour Feerik. De plus, acquérir de nouvelles fonctionnalités pas à pas fait aussi partie du modèle économique de Feerik.

Techniquement, l'interface graphique de l'éditeur de comportements est réalisée en Macromedia Flash. Les instructions de l'utilisateur sont ensuite compilées et utilisées par la plate-forme de simulation multi-agent TurtleKit [11] pour produire un fichier xml représentant le déroulement complet d'une partie. Ce fichier est ensuite utilisé par un moteur 3D pour navigateur appelé Unity [7]. La figure 6 montre le résultat obtenu au final dans le navigateur Internet de l'utilisateur.

4.3 Premiers retours utlisateurs

Malgré le fait qu'aucun graphiste de métier n'ait encore travaillé sur le design de l'interface graphique, les premiers retours que nous avons eus de la part d'utilisateurs non informaticiens sont très positifs. En effet, ceux-ci construisent rapidement des comportements agents et trouvent les principes de la PS assez intuitifs. En particulier, les utilisateurs apprécient que (1) les situations puissent être définies par de simples clics de souris et que (2) la paramétrisation des plans se fasse grâce à des formulaires de saisie identiques à ceux que l'on peut trouver sur des pages web classiques. Les utilisateurs trouvent aussi très intuitif la manière dont les situations peuvent être classées par ordre de priorité grâce à des actions de type glisser/déposer.

Le *game-play* étant un aspect majeur d'un jeu vidéo, un travail important reste à faire en ce qui concerne le retour que l'utilisateur a sur la manière dont ces robots se comportent durant la bataille. Outre le fait de gagner ou de perdre, il est en effet très important que l'utilisateur ait un moyen d'évaluer les forces et les faiblesses des comportements qu'il aura créés. Dans cette perspective, il est prévu de collecter, pendant le déroulement d'une partie, des informations statistiques telles que la nature des plans utilisés, le temps passé dans l'exécution de chaque plan, l'énergie consommée par un type de plan, etc.

5 Limitations actuelles

Comme nous l'avons dit, la PS ne prétend pas permettre à un programmeur débutant de construire un SMA à partir de rien. En effet, il est nécessaire de faire appel à un expert en informatique pour programmer un environnement dédié à un domaine précis. En particulier, cela nécessite que le modèle soit programmé en collaboration avec les experts du domaine visé. Cependant, il faut remarquer que cela est dans une certaine mesure toujours vrai, quel que soit le domaine considéré. Par ailleurs, nous abordons dans notre démarche ce problème au niveau génie logiciel, de telle manière que la structure du logiciel puisse être réutilisée facilement pour de nouveaux domaines. Nous espérons ainsi limiter l'effort de développement nécessaire à l'élaboration d'outils de PV basés sur la PS.

Une autre limitation porte sur le fait que les utilisateurs ne peuvent pas ajouter de nouveaux percepts, ni modifier la façon dont ils sont utilisés. La même remarque s'applique à la structure des plans. A notre avis, il ne s'agit pas d'une limitation définitive, en tout cas, en ce qui concerne les percepts. Par exemple, des utilisateurs pourraient avoir accès à une page d'édition qui permettrait de représenter les caractéristiques de l'environnement. L'utilisateur pourrait alors sé-

7. http ://unity3d.com/

lectionner des propriétés particulières pour définir de nouveaux percepts en quelques clics.

6 Travaux de recherche liés

Concernant la programmation de comportements complexes, un champ de recherche intéressant porte sur la programmation par imitation. Par exemple, dans le cadre de la programmation de robots footballeurs, il a été proposé dans [1] de suivre la manière dont, dans un jeu vidéo, un humain joue le rôle d'un robot. Cela afin de modéliser son comportement, grâce à des techniques d'apprentissage automatique, et ensuite utiliser ce comportement pour un robot. À l'inverse, la PS pourrait être utilisée comme une manière d'intégrer des comportements complexes sans avoir à modéliser le comportement humain, puisqu'il est entièrement intégré aux comportements résultants. Par exemple, à partir de plusieurs joueurs de Warbot, il est possible d'extraire des patterns de programmation récurrents afin de les étudier et d'analyser les plus efficaces.

Il existe sans aucun doute un lien entre la conception participative de SMA (voir par exemple [7]) et la PS du fait que le comportement de l'agent artificiel obtenu résulte en majeure partie des choix de l'utilisateur, qui matérialise ainsi sa propre manière de penser chaque situation. Cependant, ces deux formes diffèrent, à la fois en termes d'objectifs et de flux de conception. Dans un scénario participatif, l'utilisateur joue un rôle pendant la simulation et ne programme rien, ce qui est par exemple incompatible avec le mode de fonctionnement de Warbot. Au contraire, en PS, l'utilisateur doit programmer *off-line* tous les comportements et il ne peut pas intervenir pendant la simulation. Néanmoins, il est clair que ces deux formes partagent des préoccupations communes qu'il serait souhaitable d'exhiber dans l'avenir.

Enfin, l'idée d'utiliser des concepts de haut niveau pour programmer des simulations n'est pas neuve. Par exemple, dans [6], les auteurs se penchent sur l'intérêt d'un langage de haut niveau (SETL [16]) pour la simulation. Ils affirment qu'un tel langage doit incorporer des structures de données et des procédures de haut niveau. Bien que la PS s'appuie sur une telle philosophie, il propose un langage dédié à la programmation d'agents (SETL est un langage générique de manipulation d'ensembles). Néanmoins, il serait intéressant d'étudier comment les principes de conception liés à l'utilisation de langages de haut niveau peuvent être utilisés dans notre approche.

7 Conclusion

Cet article a présenté une approche de programmation orientée agent, appelée programmation situationnelle, qui permet à des utilisateurs non-informaticiens de créer des comportements d'agents sur un domaine ciblé. Plus précisément, la PS définit des principes de conception qui peuvent être utilisés par des informaticiens pour développer des plate-formes orientées PS et dédiées à des débutants en informatique ou en SMA. L'intérêt est alors de pouvoir créer des comportements complexes d'agents tout en bénéficiant d'interfaces de création extrêmement simples.

Nous avons montré comment la PS est appliquée dans le cadre d'un jeu video multi-agent en ligne. Même s'il reste encore du travail à faire sur l'interface graphique et l'esthétique de cette version de Warbot, les premiers retours que nous avons eus sont prometteurs et montrent que la PS est une solution concrète permettant à tout utilisateur d'ordinateur de programmer des comportements artificiels. Ainsi, outre le fait que la PS nous ait permis d'atteindre les objectifs fixés dans le cadre du projet Warbot, nous pensons que ce type d'approche participe à promouvoir l'utilisation et la compréhension des SMA dans la société. Ce qui constitue l'un des défis sociétaux majeurs pour les SMA.

Enfin, outre les perspectives dont nous avons parlées dans la section précédente, nous comptons maintenant enregistrer des données utilisateurs pour analyser les patterns de programmation de comportements qui seront issus de ce cas d'étude. Cela nous permettra dans un premier temps d'accroître le *game play* de Warbot en récompensant les meilleurs joueurs. Ensuite, dans une perspective plus générale, nous pensons que la PS pourrait s'avérer une alternative viable à la *conception participative* ou à l'imitation d'humains. Notamment, la programmation situationnelle pourrait aussi être utilisée pour traduire des comportements humains en programmes informatiques, afin soit (1) d'étudier leurs caractéristiques « in silico », soit (2), d'intégrer des comportements réalistes dans des SMA.

Références

[1] Ricardo Aler, Jose M. Valls, David Camacho, and Alberto Lopez. Programming Ro-

bosoccer agents by modeling human behavior. *Expert Systems with Applications*, 36(2, Part 1) :1850 – 1859, 2009.

[2] François Bousquet, Innocent Bakam, Hubert Proton, and Christophe Le Page. Cormas : Common-pool resources and multi-agent systems. In *Proceedings of the 11th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 826–837. Springer-Verlag, 1998.

[3] Paul Hsueh-Min Chang, Kuang-Tai Chen, Yu-Hung Chien, Edward Chao-Chun Kao, and Von-Wun Soo. From reality to mind : A cognitive middle layer of environment concepts for believable agents. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems, First Int. Workshop, E4MAS 2004, NY, USA, July 19, 2004, Revised Selected Papers*, volume 3374 of *LNCS*, pages 57–73. Springer, 2005.

[4] Jacques Ferber. *Les systèmes multi-agents, vers une intelligence collective*. Interéditions, 1995.

[5] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From agents to organizations : an organizational view of multi-agent systems. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *Agent-Oriented Software Engineering IV*, volume 2935 of *LNCS*, pages 214–230. Springer Verlag, January 2004.

[6] W. R. Franta and Kurt Maly. The suitability of a very high level language (setl) for simulation structuring and control. In Bernard Robinet, editor, *Symposium on Programming*, volume 19 of *LNCS*, pages 156–169. Springer, 1974.

[7] Paul Guyot and Shinichi Honiden. Agent-based participatory simulations : Merging multi-agent systems and role-playing games. *Journal of Artificial Societies and Social Simulation*, 9(4) :8, 2006.

[8] Andri Ioannidou, Alexander Repenning, and David C. Webb. AgentCubes : Incremental 3d end-user development. *Journal of Visual Language & Computing*, 20(4) :236–251, 2009.

[9] F. Klügl, R. Herrler, and M. Fehler. SeSAm : implementation of agent-based simulation using visual programming. In *AAMAS '06 : Proceedings of the fifth international joint conference on Autonomous Agents and Multiagent Systems*, pages 1439–1440, NY, USA, 2006. ACM.

[10] Franziska Klügl and Ana L. C. Bazzan. Route decision behaviour in a commuting scenario : Simple heuristics adaptation and effect of traffic forecast. *JASSS, The Journal of Artificial Societies and Social Simulation*, 7(1), 2004.

[11] Fabien Michel, Grégory Beurier, and Jacques Ferber. The TurtleKit simulation platform : Application to complex systems. In Alain Akono, Emmanuel Tonyé, Albert Dipanda, and Kokou Yétongnon, editors, *Workshops Sessions, First International Conference on Signal & Image Technology and Internet-Based Systems SITIS' 05*, pages 122–128. IEEE, november 2005.

[12] Fabien Michel, Jacques Ferber, and Alexis Drogoul. Multi-Agent Systems and Simulation : a Survey From the Agents Community's Perspective. In Danny Weyns and Adelinde Uhrmacher, editors, *Multi-Agent Systems : Simulation and Applications*, Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 3–52. CRC Press-Taylor & Francis, 06 2009.

[13] Juan Pavón, Candelaria Sansores, and Jorge J. Gómez-Sanz. Modelling and simulation of social systems with INGENIAS. *Int. J. Agent-Oriented Softw. Eng.*, 2(2) :196–221, 2008.

[14] Alexander Repenning, Andri Ioannidou, and John Zola. AgentSheets : End-user programmable simulations. *Journal of Artificial Societies and Social Simulation, JASSS*, 3(3), 2000.

[15] Candelaria Sansores, Juan Pavón, and Jorge J. Gómez-Sanz. Visual modeling for complex agent-based simulation systems. In Jaime Simão Sichman and Luis Antunes, editors, *Multi-Agent-Based Simulation VI, International Workshop, MABS 2005, Utrecht, The Netherlands, July 25, 2005, Revised and Invited Papers*, volume 3891 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 2006.

[16] J Schwartz. *Set Theory as a Language for Program Specification and Programming*. Courant Institute of Mathematical Sciences, New York University, 1970.

[17] Toby Tyrrell. The use of hierarchies for action selection. *Adaptive Behavior*, 1(4) :387–420, 1993.