UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

UNIVERSITÉ DE FRANCHE-COMTÉ

SHINE research

THÈSE DE DOCTORAT DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

École doctorale n°37

Sciences Pour l'Ingénieur et Microtechniques

Doctorat d'Informatique

par

DIOT NICOLAS

SAMP : Plateforme de modélisation à partir du paradigme multi-agents pour l'univers du jeu vidéo : vers un développement accessible et une gestion adaptée des interactions

Thèse présentée et soutenue à Besançon, le 19 décembre 2018

Composition du Jury :

MANDIAU RENÉ	Professeur des universités à l'Université de Valenciennes	Président
MICHEL FABIEN	Maître de conférences HDR à l'Université de Montpellier	Rapporteur
AMBLARD FRÉDERIC	Professeur des universités à l'Université de Toulouse 1	Rapporteur
BOUQUET FABRICE	Professeur des universités à l'Université de Franche-Comté	Directeur de thèse
LANG CHRISTOPHE	Maître de conférences à l'Université de Franche-Comté	Codirecteur de thèse
GROSDEMOUGE SYLVAIN	Président de Shine Research	Référent entreprise

école doctorale **sciences pour l'ingénieur et microtechniques**

Titre : SAMP : Plateforme de modélisation à partir du paradigme multi-agents pour l'univers du jeu vidéo : vers un développement accessible et une gestion adaptée des interactions

Mots-clés : Jeux vidéo, système multi-agents, interaction, accessibilité, Shine Agent Modeling Platform

Résumé :

En quelques années, les domaines des jeux vidéo et des systèmes multi-agents (SMA) ont connu un très bel essor. Malgré des similitudes assez fortes entre les deux domaines (présence d'entités dans les jeux vidéo pouvant être assimilées à des agents), les SMA ne sont presque jamais utilisés dans le développement de jeux. Ce mémoire présente Shine Agent Modeling Platform (SAMP), une plateforme visant à intégrer le paradigme multi-agents au sein du développement de jeux vidéo. Cette fusion permet l'utilisation de la puissance des multi-agents au sein des jeux vidéo.

SAMP propose une approche au niveau des interactions permettant de réduire le coût de traitement de ces interactions en optimisant le nombre de recherches effectuées dans l'environnement.

En plus d'intégrer le paradigme multi-agents, SAMP vise à être accessible à un maximum d'utilisateurs en proposant une interface de modélisation entièrement graphique. Un système d'importation de modèles comportementaux permet de créer deux niveaux de modélisation : un niveau proche de la logique développement informatique et un niveau proche de la logique métier de l'utilisateur.

SAMP est intégré à un moteur de jeux vidéo, Shine Engine, permettant de générer les environnements graphiques dans lesquels les agents évolueront.

Title: SAMP : Plateforme de modélisation à partir du paradigme multi-agents pour l'univers du jeu vidéo : vers un développement accessible et une gestion adaptée des interactions

Keywords: Video games, multi-agents system, interaction, user-friendly, Shine Agent Modeling Platform

Abstract:

In recent years, video games and multi-agents systems (MAS) domains has become more and more present. Despite of strong similarities (video games entities which can be assimilated to agents), MAS are very rarely used during the development of video games. This thesis presents the Shine Agent Modeling Platform (SAMP), a framework trying to integrate the multi-agents paradigm within the development of video games. The purpose is to integrate the efficiency of the MAS within the video games.

SAMP provides an approach to enhance the interactions between agents. This approach reduces the number of searches within the environment.

In addition, to integrate the multi-agents paradigm within the video games, SAMP aims to be user-friendly by proposing a full graphical interface to MAS. An import/export system of these models allows users to create two modeling levels: one close to the computer sciences logic and the second close the business logic of the user.

SAMP is integrated in a video games engine: Shine Engine. This integration allows to generate the graphic environment in which agents will live.

UBFC UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 BesanÃğon, France

A Christian qui était toujours si fier de moi et à qui je pense si souvent,

REMERCIEMENTS

Avant toute chose, je tiens à remercier toutes les personnes qui ont, de près ou de loin participé à cette aventure avec moi.

Merci Fabrice et Christophe pour votre accompagnement, votre patience, vos conseils et votre humour constant qui m'ont permis de ne pas voir passer ces années de recherches. Merci au laboratoire FEMTO-ST et son département DISC de l'université de Franche-Comté de m'avoir accueilli.

Merci à l'entreprise Shine Research qui m'a accueilli tout au long de cette thèse et m'a permis de travailler dans un domaine pour lequel j'ai toujours eu une forte attirance. Merci Sylvain pour l'accompagnement qui a été le tien tout au long de mes travaux de recherche. Ton expertise du domaine des jeux vidéo m'a guidé dans beaucoup de choix technologiques. Merci à Bastien, Rémy, Johann et Thomas qui ont été là depuis le début et ont toujours su m'accompagner... à leur manière. Petite pensée pour Stéphane tout récemment arrivé et à qui je dois annoncer que j'ai perdu.

Merci à mes parents, mes beaux-parents et surtout à ma femme et ma fille. Ces derniers jours de rédaction n'ont pas été les plus simples pour moi, mais aussi pour vous. Merci pour votre soutien et votre aide sans failles.

Merci à Carnat de toujours avoir su trouver mes blessures et mes faiblesses, ce que j'avoue qu'à demi-mots. Je ne remercie pas celui qui pensait que mes encadrants voulaient que *je me taise et non que je fasse une thèse*. Il se reconnaitra.

Un grand merci aux relecteurs/correcteurs. Relire une thèse lorsque l'on ne comprend pas grand-chose à son contenu a dû être un travail de longue haleine. S'il reste des fautes, je ne vous en tiendrai pas rigueur.

Merci à Messieurs Fabien Michel et Frédéric Amblard pour leur travail de rapporteurs sur ce mémoire et à Monsieur René Mandiau d'avoir accepté de présider le jury de soutenance de ma thèse. J'espère que la lecture et la soutenance vous seront agréables.

SOMMAIRE

I

CONTEXTE ET PROBLÉMATIQUES

1

INTRODUCTION

Sommaire

1.1/ CONTEXTE

Le contexte est celui des jeux vidéo mais aussi celui des systèmes multi-agents. L'industrie du jeu vidéo est en constante évolution pour combler les attentes du public et proposer de nouvelles expériences ludiques. L'évolution des périphériques a permis de repenser et d'améliorer le jeu vidéo. Au commencement, on retrouve des jeux simplistes tel que PONG (créé par la société Atari) qui voyait deux joueurs s'affronter dans un jeu de raquette minimaliste ou Core War (créé par la société Bell) qui faisait s'affronter des programmes reproducteurs entre eux. C'est l'arrivée des bornes d'arcades qui a permis de démocratiser le jeu vidéo auprès du grand public.

En 1978, Taito sort le célèbre Space Invaders suivi par Atari qui livre, en 1979, Asteroid et fait entrer le jeu vidéo dans ce que certains nomment l'*âge d'or de l'arcade*. Le jeu vidéo entre aussi dans les foyers, d'abord sur consoles de jeux, des périphériques dédiés aux jeux vidéo, puis sur micro-ordinateur.

S'ensuit une évolution constante de l'industrie du jeu vidéo, proposant à chaque génération de jeux de nouvelles manières de jouer. En 2014, le revenu mondial du jeu vidéo est estimé à 81 milliards de dollars (le double de ce que l'industrie du cinéma a généré l'année d'avant).

Les avancées technologiques ont permis le développement de jeux plus complexes, où l'expérience de jeu est immersive. Par exemple, le comportement des personnages non-joueurs (PNJ) dans un jeu a beaucoup évolué depuis les débuts du jeu vidéo. Si on prend l'exemple de Space Invaders, les ennemis se déplaçaient d'un bord à l'autre de l'écran, et lorsqu'un bord était atteint, ils descendaient d'un cran sur l'écran et repartaient dans la direction opposée. On peut aussi citer des classiques comme Mario ou Castlevania dans lesquels les ennemis suivent ce qu'on appelle des patterns [1] (cf figure 1.1). Dans cette

1. Un pattern est un comportement qu'un ennemi dans un jeu vidéo va suivre. Chaque ennemi d'un même type suit le même pattern ce qui permet au joueur qui a mémorisé ces patterns d'avancer plus facilement dans le jeu.

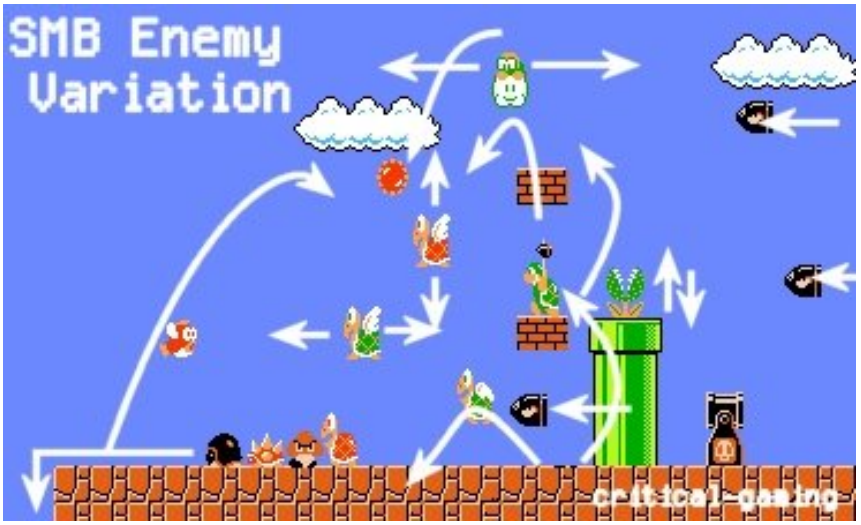

FIGURE 1.1 – Le pattern des ennemis dans le jeu New Super Mario Bros.

figure, chaque flèche représente le mouvement que l'ennemi suivra tout le temps.

Lorsque nous décrivons le comportement des PNJ dans un jeu, il est facile de faire le parallèle avec le paradigme multi-agents. En effet, si nous considérons chaque PNJ comme un agent, le jeu vidéo devient alors un système multi-agents (SMA). C'est naturellement que nos recherches se sont dirigées dans cette direction. Cependant, même si certains jeux vidéo semblent être développés en se basant sur le paradigme multi-agents, ils n'en appliquent pas toujours les règles et ne font qu'adapter certains aspects du paradigme.

Aujourd'hui, les jeux proposent des ennemis (ou des alliés) avec des comportements plus complexes. Certains ennemis sont, par exemple, capables de se coordonner pour prendre à revers le joueur qui se serait mis à couvert derrière un obstacle. Ces comportements pourraient se rapprocher de comportements intelligents mais, bien souvent, il ne s'agit que de comportements réactifs, où les agents ne font que répondre d'une manière particulière en fonction de l'environnement qui les entoure.

La modélisation de ces comportements est un sujet prégnant dans le monde du jeu vidéo.

Ces comportements permettent de favoriser l'immersion du joueur et, par conséquent, son expérience de jeu. Le *framerate*, le nombre d'images par seconde (ou Frame Per Second (**FPS**)), a aussi une incidence sur l'expérience de jeu. Avec un framerate trop bas, les images s'enchaineront trop lentement et le jeu sera saccadé à l'écran (fluidité visuelle). Si au contraire le framerate est élevé, il se peut que les entités du jeu se mettent à jour trop peu souvent (fluidité intellectuelle) ce qui a aussi un impact sur l'expérience de jeu. Le framerate est directement influencé par le fait que, aujourd'hui, les jeux vidéo mettent en scène de plus en plus d'agents et que ces agents ont un comportement de plus en plus complexe. Cela entraîne une augmentation de la consommation des ressources pouvant nuire au temps de calcul et donc au framerate.

Il est alors nécessaire d'optimiser l'exécution des jeux, notamment pour tenir compte de cette contrainte.

Un autre point intéressant dans le monde du jeu vidéo est l'apparition, il y a quelques années, sur le marché, de développeurs indépendants. Un des premiers succès dans

ce domaine est le jeu Minecraft, développé par Markus Persson (aka Notch). Suite au succès de ce jeu, beaucoup de développeurs amateurs ont essayé avec plus ou moins de réussite de développer leurs propres jeux. De plus, les équipes de développement de studios professionnels sont composées d'acteurs différents : développeurs, designers, graphistes, animateurs, etc. Cette hétérogénéité d'acteurs, chacun avec leurs compétences, nous conforte dans le besoin de rendre notre solution accessible aux néophytes en développement informatique.

Il nous semble alors pertinent, dans la mise en œuvre d'une solution de développement de jeu vidéo, de prendre en compte le niveau hétérogène des utilisateurs.

1.2/ PROBLÉMATIQUES

Nous nous plaçons donc dans un contexte où le jeu vidéo est un système multi-agents avec des obligations d'exécution en temps réel et d'intégration de moteurs physiques, inhérents aux jeux vidéo. Il faut aussi bien garder à l'esprit que les jeux vidéo contiennent aujourd'hui des environnements contenant des milliers d'agents et que ces agents sont capables de communiquer et se coordonner.

Sachant que la coordination entre différents agents se fait à l'aide d'interactions (entre l'agent et son environnement ou entre les agents eux-mêmes) et que le nombre de ces interactions est conséquent dans bon nombre de jeux vidéo, nous pensons que réduire ces interactions permettrait de répondre à la contrainte de performance. Mais comment réduire l'impact de ces interactions sur le système ? Comment permettre une intégration complète des moteurs physiques dans le fonctionnement des SMA développés ? Comment permettre une intégration complète des moteurs physiques dans le fonctionnement des SMA développés ?

Dans un autre registre, comme l'a fait remarquer Yoann Kubera [52], un des principaux problèmes est que les SMA sont modélisés par les spécialistes du domaine et qu'ils sont implémentés par des développeurs en informatique. Et cette situation impose qu'il y ait une traduction pour *passer du modèle à l'implémentation* durant laquelle il y a un risque que des informations soient perdues ou mal traduites.

De par l'émergence de ces différents profils de créateurs de jeux vidéo et l'hétérogénéité des compétences de chacun, nous avons cherché à répondre à une problématique de complexité. Comment rendre notre outil simple d'utilisation tout en permettant une grande expressivité dans le développement ? Comment permettre de développer tous les comportements possibles tout en permettant à un néophyte en développement de réaliser ces comportements ?

Afin de répondre à toutes ces questions, nous présentons, dans cette thèse nos travaux et plus particulièrement SAMP (Shine Agent Modeling Platform), une approche permettant de **développer un jeu vidéo à base de SMA** en **s'affranchissant de l'apprentissage d'un langage informatique**. SAMP est décomposée en 3 parties (figure 1.2) : une partie définissant le méta-modèle de SAMP (SAMP-M), une partie d'édition de modèles et de génération des SMA (SAMP-E) et une partie permettant la gestion de l'exécution des SMA (SAMP-X). Le tout offrant des performances permettant de **jouer en respectant nos contraintes de temps réel**. Enfin, le besoin d'intégrer, aux SMA développés à l'aide de SAMP, la capacité d'utiliser des outils pour simuler les principes de la physique appelés

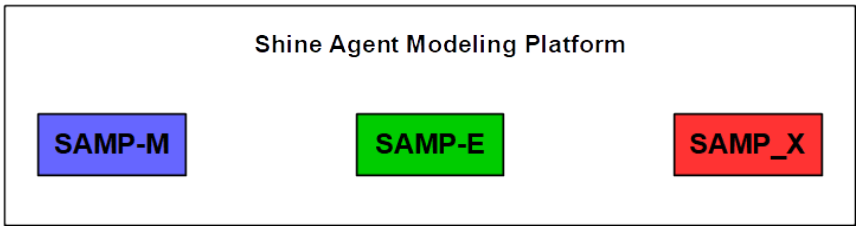

FIGURE 1.2 – Représentation des 3 parties de SAMP

moteurs physiques.

C'est en prenant en compte toutes ces composantes propres au développement d'un jeu vidéo que nous avons orienté nos travaux de recherche.

1.3/ PLAN MÉMOIRE

La suite de ce mémoire est découpée ainsi :

- Le chapitre 2 réalise un état de l'art. Nous abordons d'abord le concept de jeux vidéo que nous définissons et dont nous introduisons les deux principaux outils qui sont les moteurs de jeux et les moteurs physiques. Nous continuons par un état de l'art du paradigme multi-agents en abordant les agents, les SMA, les environnements et les communications. Nous enchaînons par un état de l'art sur le temps réel en SMA afin de respecter le principe de frame propre au domaine du jeu vidéo. Avant de conclure ce chapitre nous faisons un état de l'art des outils de modélisation de comportement, que ce soit en SMA ou dans le domaine du jeu vidéo ;
- Le chapitre 3 présente un exemple d'un système proies-prédateurs qui servira à illustrer les explications de nos contributions ;
- Dans le chapitre 4 nous décrivons Shine Agent Modeling Platform en commençant par décrire ses concepts fondamentaux. Nous décrivons ensuite les éléments qui le composent : les compétences, les interactions, les agents et la modélisation des comportements ;
- Nous définissons ensuite, dans le chapitre 5, une nouvelle approche pour les interactions basée sur un principe d'agents actifs et passifs ;
- Le chapitre 6 explique comment, à partir de la modélisation du système nous générons le code qui, une fois compilé, générera un *plugin* compatible avec les jeux vidéo développés à l'aide de Shine Engine ;
- Nous évaluons ensuite, dans le chapitre 7, la méthode, d'abord d'un point de vue théorique. Dans ce chapitre, nous abordons comment nous avons obtenu les résultats théoriques concernant la comparaison de notre approche des interactions avec une approche classique ;
- Nous exposons enfin les résultats de nos expérimentations dans le chapitre 8. Cela concerne, dans un premier temps, des expérimentations permettant de comparer notre approche des interactions avec une approche classique. Nous nous focalisons ensuite sur la simplicité d'utilisation et l'expressivité de SAMP.
- Nous terminons par une conclusion et une discussion sur les travaux décrits dans ce mémoire dans le chapitre 9.

2

ÉTAT DE L'ART

Sommaire

Les systèmes multi-agents et les agents qui les composent peuvent être définis et utilisés de différentes manières. Cela dépend d'abord du domaine dans lequel nous cherchons à utiliser ces agents et de l'objectif à atteindre. Un SMA utilisé pour effectuer une simulation d'un environnement naturel n'a pas les mêmes restrictions et besoins qu'un SMA utilisé dans le but de contrôler plusieurs robots et de les faire se coordonner.

Le type des agents utilisés, l'environnement dans lequel ils évoluent ou leur mode de communication sont différents points qu'il est important de définir avant le développement d'un SMA afin de répondre au mieux aux objectifs fixés. Comme nous l'avons indiqué, nos travaux prennent place dans le domaine des jeux vidéo. C'est un domaine en perpétuelle évolution et qu'il est important de définir afin de pouvoir au mieux appréhender la suite de ce mémoire.

Dans ce chapitre, nous commençons par définir les spécificités de notre contexte applicatif qui est le domaine des jeux vidéo en explicitant le lien entre celui-ci et les SMA. Nous présentons, ensuite, les concepts et les approches associés à la réalisation d'un SMA. Nous nous attardons, enfin, sur les recherches abordant le temps réel dans les SMA.

2.1/ LES JEUX VIDÉO

Dans cette section, nous commençons par décrire ce qu'est un jeu vidéo et ce qui le compose. Pour cela, nous présentons ce que sont les moteurs physiques et comment fonctionne le moteur d'un jeu. Nous nous attardons sur le moteur Shine Engine dans lequel SAMP a été intégré en tant que plugin. Nous terminons par expliquer le fonctionnement interne propre à une grande majorité de jeu.

2.1.1/ QU'EST CE QU'UN JEU VIDÉO

Le domaine des jeux vidéo est vaste et varié. Chaque année des milliers de nouveaux jeux voient le jour sur des supports très différents les uns des autres : consoles de jeux, ordinateurs, appareils mobiles.

Il existe de nombreux genres de jeux différents avec des budgets et des équipes toutes aussi différentes. On peut parler de jeu AAA [1] comme *Far Cry* ou *Battlefield*, de jeux vidéo *casual* comme *Angry Birds* ou *Candy Crush Saga* ou de productions indépendantes comme *Minecraft*, toutefois tous les jeux vidéo sont caractérisés par les mêmes propriétés.

Nous définissons un jeu vidéo comme :

— devant être exécuté sur un dispositif numérique ;

— devant être ludique ;

— devant permettre des interactions de la part de son ou ses utilisateurs, par le biais de périphériques : manettes, clavier, souris, écran tactile, détecteur de mouvement, capteurs de position, etc ;

— devant permettre aux actions des utilisateurs d'avoir un impact sur le jeu et que ces impacts puissent être perçus par les utilisateurs : affichage sur un écran ou dans un casque, retour haptique, son, etc.

Le domaine des jeux vidéo est un domaine en constante évolution. De nombreuses avancées technologiques sont réalisées par les industries du jeu vidéo, notamment en ce qui concerne les algorithmes de rendu graphique. En 2018, Nvidia met en avant un algorithme permettant d'afficher une scène en temps réel en utilisant le Ray Tracing. Le Ray Tracing vise à reproduire le plus fidèlement le rendu de la lumière dans une scène 3D.

Les avancées technologiques ne se contentent pas du seul rendu visuel. De nombreuses technologies ont été adoptées et grandement améliorées par l'industrie du jeu vidéo. La démocratisation des *smartphones* et l'utilisation grandissante de ces terminaux pour le jeu a amené de nombreuses avancées technologiques notamment dans la miniaturisation des composants, l'amélioration de l'autonomie des batteries ou l'efficacité des écrans tactiles. La détection de mouvement développée par Microsoft pour la Xbox 360 avec Kinect est aujourd'hui utilisée dans les domaines de l'industrie (amélioration des postures de travail des ouvriers) ou dans les domaines médicaux (pilotage de terminaux informatiques par le mouvement dans les blocs opératoires).

1. AAA - ou triple-A - est une classification de l'industrie du jeu vidéo désignant des jeux dont les budgets de développement et de promotion sont élevés et dont les évaluations par les critiques professionnels sont bonnes.

Tous les jeux vidéo n'ont pas les mêmes objectifs : Certains sont développés dans le but de distraire leurs utilisateurs. Ils amènent une réflexion artistique ou sociale en faisant intervenir différentes émotions chez les joueurs. Certains autres jeux ont une portée pédagogique, médicale ou scientifique voir publicitaire. Ces jeux sont appelés jeux sérieux (serious game), du terme *Serio Ludere* issu du mouvement humaniste italien du XV^{e} siècle qui visait à aborder un sujet sérieux de manière ludique.

Les jeux vidéos font intervenir un grand nombre de métiers. Dans le développement d'un jeu vidéo peuvent intervenir des développeurs en informatique, des graphistes, des modélisateurs (pour la création des squelettes des objets 3D), des animateurs (pour l'animation des objets 3D ou 2D), des ingénieurs du son, des concepteurs de niveaux, des concepteurs de règles du jeu, des scénaristes, etc. Parfois, des métiers qui n'ont aucun lien avec le jeu vidéo sont mis à contribution. Par exemple, lors du développement des jeu *Battlefield 1* et *Call of Duty : World War II*, des historiens ont travaillé sur l'aspect historique des jeux.

Comme dit précédemment les *serious games* sont utilisés dans de nombreux domaines de recherche où ils sont utilisés comme un support de travail [85]. Certains de ces travaux visent à aider les patients atteints de troubles autistiques[2] [14, 43]) ou encore visent à développer des outils pour l'apprentissage [47, 15].

Maintenant que nous avons défini ce qu'est un jeu vidéo et abordé tout ce qui gravite autour de ce domaine, nous allons expliquer ce qui compose un jeu vidéo que ce soit pour la gestion et l'affichage des informations ou pour le fonctionnement du monde qu'il émule.

2.1.2/ LES MOTEURS PHYSIQUES

Depuis des dizaines d'années, des outils permettant la gestion de la physique dans les jeux vidéo ont été développés. Il s'agit des moteurs physiques [19, 65]. Ces moteurs permettent de calculer des effets physiques : gravité, forces de frottements, jointures, etc. Presque chaque jeu vidéo aujourd'hui utilise un moteur physique, du basique Angry Birds au plus évolué Battlefield. Les moteurs ont sans cesse évolué pour améliorer leur fonctionnement et assurer aujourd'hui de façon efficace le réalisme des effets et l'optimisation des calculs (notamment en réalisant leur calcul directement sur GPU).

Le moteur physique a un fonctionnement simple : on crée un monde que l'on paramètre avec ses différentes caractéristiques (gravité, densité de l'air, dimensions). Ensuite, on crée des entités physiques que l'on paramètre (masse, friction, jointure, etc) et que l'on ajoute au monde. Il sera alors possible d'appliquer des forces aux entités pour qu'à chaque boucle, le moteur physique calcule le nouvel état de chaque entité.

L'utilisation d'un moteur physique dans un jeu vidéo est une nécessité pour des raisons économiques (pas de besoin de réécrire du code) et pour des raisons de performances (les moteurs physiques sont de plus en plus évolués). Ainsi, il est nécessaire que l'outil que nous voulons développer puisse utiliser la puissance des moteurs physiques. Nous allons maintenant aborder un nouvel outil du jeu vidéo : les moteurs de jeux vidéo.

2. Selon la classification internationale des maladies de l'OMS (CIM 10), l'autisme est un trouble envahissant du développement qui affecte les fonctions cérébrales. Il n'est plus considéré comme une affection psychologique ni comme une maladie psychiatrique.

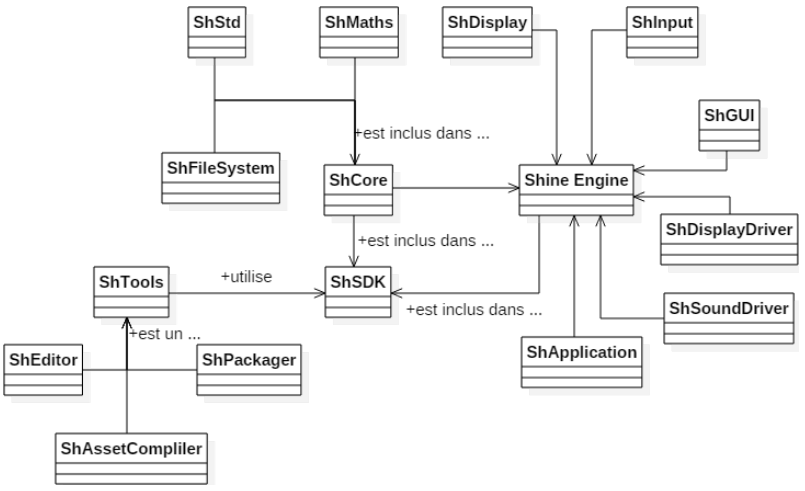

FIGURE 2.1 – Diagramme représentant les modules les plus importants de Shine Engine

2.1.3/ LES MOTEURS DE JEUX VIDÉO

Les moteurs physiques permettent l'intégration rapide et efficace des principes de la physique dans les jeux vidéo. Il est un autre outil qui est devenu indispensable au développement d'un jeu vidéo : le moteur de jeu. Là où le moteur physique permet de gérer tout ce qui a trait à la physique des corps, le moteur de jeu va gérer tout ou partie du fonctionnement d'un jeu.

Les moteurs de jeux vidéo permettent de gérer l'affichage des entités, l'activation des sons, les calculs de luminosité, les entrées utilisateurs, mais aussi d'interfacer différentes API externes (réseaux sociaux, discussions, store pour achat complémentaire, connexion aux serveurs de jeu, etc). La figure 2.1 montrent les différents modules composant le moteur Shine Engine. Bien qu'utilisant pour exemple Shine Engine, ces modules sont aussi présents dans d'autres moteurs de jeux vidéo.

Un des avantages majeurs de l'utilisation des moteurs de jeux vidéo est la diminution des coûts de développement. Il existe un grand nombre de moteurs de jeux vidéo possédant, pour une grande majorité, un éditeur graphique. On peut citer parmi les plus connus Unreal Engine, Unity, Game Maker ou Rpg maker. Chacun permet le développement de jeux vidéo plus ou moins complexes de manière plus ou moins fermée.

Rpg Maker est spécialisé dans le développement de jeux de type Rpg (Role Playing Game ou Jeu de rôle). Game Maker est lui spécialisé dans le développement de jeu 2D. Ces deux éditeurs sont très simples d'utilisation est permettent le développement de jeux vidéo sans avoir recours à l'utilisation de langage de programmation textuel (comme du C++ ou du Java). Il est cependant possible de développer des parties de jeux à l'aide de code textuel afin de créer des fonctionnalités non prises en compte par l'un de ces éditeurs.

Unreal Engine et Unity sont des moteurs possédant des capacités et des outils plus complets que Rpg Maker ou Game Maker. Les possibilités de développement sont plus grandes avec ces deux moteurs de jeux, mais sont aussi plus compliquées à prendre en main. Ils proposent cependant un système de programmation graphique permettant de créer des jeux sans avoir recours à un langage de programmation textuelle.

L'entreprise dans laquelle cette thèse a été développée possède un moteur de jeux vidéo appelé Shine Engine [3] développé par l'entreprise Shine Research [4]. Ce moteur multi-plateformes est capable de générer un jeu compatible pour différentes plateformes à partir d'un seul code source. Shine Engine gère l'intégration de moteur physique par le biais de l'utilisation de *plugins*. La figure 2.1 montre les différents modules composant Shine Engine.

On constate dans cette figure 2.1 qu'il y a 3 groupes de modules :

- *Shine Engine* qui contient toutes les bibliothèques multi-plateformes du moteur. Tous ces modules sont multi-plateformes.
  - *ShStd*, *ShMaths* et *ShFileSystem* redéfinissent des méthodes des bibliothèques standard, maths et système de fichiers pour les différentes plateformes ;
  - *ShInput* gère les entrées utilisateurs en utilisant différents drivers en fonction de la plateforme et des périphériques utilisés ;
  - *ShSoundDriver* et *ShDisplayDriver* permettent la communication avec les périphériques de sortie pour le son et l'affichage ;
  - *ShDisplay* gère le rendu des objets dans les scènes 3D ; *ShGUI* gère le rendu de l'interface utilisateur.
- *ShSDK* est l'API permettant d'accéder aux méthodes présentes dans *Shine Engine* ;
- Les *Tools* sont tous les outils permettant de développer du contenu dans *Shine Engine* :
  - *ShAssetCompiler* permet la compilation des ressources (textures, objet 3D, animation, fonts, ...) dans un format compatible avec *Shine Engine* ;
  - L'éditeur permet la création de différents contenus pour *Shine Engine* : niveaux 3D/2D, méta-animation, système de particules, interface utilisateur, etc. Et permet le paramétrage de nombreux éléments tels que les entrées utilisateurs (utilisées par ShInput), les ressources (sons, images, shaders, etc) ou les trophées [5] que les joueurs peuvent débloquer ;
  - *ShPackager* permet la création d'un package prêt à l'emploi pour être envoyé sur les différentes boutiques de ventes de jeux des plateformes cibles (Steam sur PC, Google Play sur Androïd, Ps Store sur Playstation, etc).

C'est au sein de cet éditeur que SAMP est développé. Il est entièrement développé en tant que plugin pour être intégré dans celui-ci.

Les moteurs de jeux vidéo sont tous différents, cependant, ils ont tous le même fonctionnement pour ce qui est de la boucle de jeu.

La boucle de jeu est un terme désignant le cheminement des instructions exécuté entre deux affichages d'un jeu vidéo. La boucle de jeu peut-être créée de différentes manières qui ont été classifiées par Luis Valente, Aura Conci et Bruno Feijòl [83]. Shine Engine utilise un de ces pattern : le pattern *Thread Unique Model Non Couplé*. La figure 2.2 expose le fonctionnement de ce pattern. On constate que dans ce pattern la boucle de

3. http ://www.shine-engine.com/
4. http ://www.shine-research.com/
5. Les trophées sont acquis lorsque les joueurs réalisent certaines actions dans les jeux. Ils permettent aux joueurs de comparer leurs progressions avec leurs amis.

jeu débute par le calcul du *delta time* correspondant au temps écoulé depuis le début de la boucle précédente. Cette valeur permet de synchroniser sur le temps certains calculs même si les boucles de jeu ont des temps d'exécution différents. Ensuite, le pattern possède une phase de récupération des entrées utilisateur puis une phase de mise à jour du moteur (par exemple la synchronisation de la position des objets dans le moteur par rapport aux entités dans le moteur physique). Dans Shine Engine, ces deux phases sont regroupées dans une phase plus complexe appelée *PreUpdate*.

Dans le pattern vient ensuite une phase de rendu. C'est durant cette phase que sont réalisés les calculs de lumières, d'ombrages et de réflexions.

La différence entre Shine Engine et ce pattern est que dans Shine Engine, après le rendu, vient une phase de *PostUpdate* dans laquelle sont réalisés certains calculs comme la mise à jour du moteur physique ou la validation des trophées.

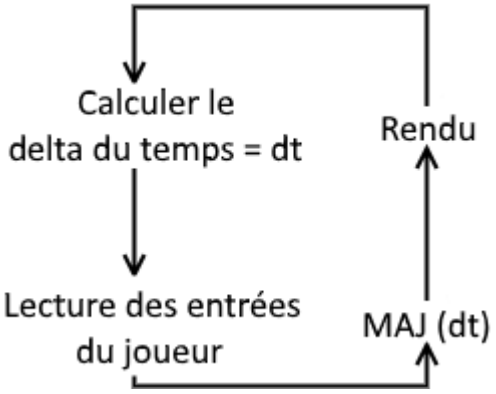

FIGURE 2.2 – Fonctionnement d'une boucle de jeu.

Nous avons défini ce que sont les jeux vidéo et les principaux outils utilisés lors de leurs développements. Nous abordons dans la suite de cette section la définition des SMA et des éléments les composants.

2.2/ LES AGENTS ET LES SYSTÈMES MULTI-AGENTS

Le paradigme multi-agents est utilisé dans divers domaines de la science, de l'industrie et du divertissement. Ce paradigme peut-être utilisé de différentes manières afin de répondre à des besoins divers. Il est possible d'utiliser des SMA afin de simuler des environnements en ayant au préalable défini les règles qui régissent cet environnement.

Il est possible d'exécuter des simulations afin d'étudier l'évolution d'une population, l'évacuation d'un bâtiment, la propagation d'un incendie, etc. Il est aussi possible d'utiliser le paradigme multi-agents afin de gérer le comportement de machines dans un environnement réel où chaque robot et chaque capteur sera un agent.

Dans le divertissement, il est possible de modéliser des scènes contenant une multitude d'agents que ce soit pour des films ou dans des jeux vidéos.

Nos objectifs sont d'utiliser la puissance des systèmes multi-agents au sein des jeux vidéo afin de pouvoir utiliser la puissance des algorithmes développés depuis une trentaine d'années dans ce domaine.

Afin de répondre au mieux à nos objectifs, nous avons cherché quels pourraient être les meilleurs choix à mettre en place dans le développement de SMA dans le domaine des jeux vidéo. Cette section contient les résultats de nos recherches. Nous commençons en donnant différentes définitions des agents, nous continuons en décrivant différents protocoles de communication entre les agents. Nous continuons en décrivant les différents rôles que peut prendre un environnement et nous terminons par la définition des SMA.

2.2.1/ LES AGENTS

Il existe un grand nombre de définitions pour un agent. Elles ont beaucoup de similitudes, mais diffèrent cependant selon le domaine dans lequel l'agent est utilisé : la robotique, l'intelligence artificielle, les études biologiques ou sociologiques. En 1971, Richard Fikes et Nils Nilsson [34] décrivent un système de résolution de problème basé sur ce qu'ils appellent des opérateurs. Dans leurs perspectives d'avenir, ils imaginent des systèmes où plusieurs opérateurs travaillent de concert. Durant leur discussion, ils imaginent aussi qu'un opérateur puisse être capable d'apprendre en fonction des tâches qu'il a déjà exécutées par le passé.

Stuart Russell et Peter Norvig [69], en 1995 (soit 25 ans plus tard), donnent cette définition de l'agent : *"Un agent est ce qui peut être vu comme percevant son environnement à l'aide de senseurs et agissant sur lui à l'aide d'effecteurs, en toute autonomie"*. Cette même année, Jacques Ferber dans [32] a donné une des premières définitions d'un agent par induction en fonction de ces aptitudes :

"On appelle agent une entité physique ou virtuelle :

- qui est capable d'agir dans un environnement ;
- qui peut communiquer directement avec d'autres agents ;
- qui est mue par un ensemble de tendances (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser) :
- qui possède des ressources propres ;
- qui est capable de percevoir (mais de manière limitée) son environnement ;
- qui ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune) ;
- qui possède des compétences et offre des services ;
- qui peut éventuellement se reproduire ;
- dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit".

Les définitions que Jacques Ferber donne sont, pour nous, les plus abouties. Sa définition ne prend pas en compte que l'agent et ce qu'il peut faire, mais aussi l'environnement dans lequel il évolue et son lien avec cet environnement ou encore la possibilité pour les agents d'avoir des compétences et des objectifs. La représentation partielle qu'un agent a de son environnement permet de s'approcher d'un comportement anthropomorphique du fait que, comme un humain, un agent possède une connaissance partielle de son environnement. Il aborde aussi le concept des compétences des agents, concept qui se révèle intéressant pour le domaine dans lequel nos recherches se basent. Dans les jeux

vidéos, l'apprentissage de capacités (savoir-faire, connaissances, compétences, ...) est souvent une partie importante de la jouabilité [6].

Mais bien que ces définitions soient les plus abouties, il est apparu au fil des années de recherche dans le domaine multi-agents deux visions différentes des agents. Rodney Brooks [21] distingue principalement deux types d'agents : les agents cognitifs issus de l'IA symbolique, et les agents réactifs représentés par la vie artificielle et l'intelligence "sans représentation". Cette distinction n'est plus si nette, les agents actuellement développés sont souvent entre ces deux bornes extrêmes. Nous plaçons ainsi le curseur en fonction des besoins du système. Nous constatons que, souvent, les systèmes contenant de très nombreux agents privilégient des entités assez basiques dans leur raisonnement.

Les agents cognitifs sont capables de prendre des décisions en fonction de leurs connaissances (environnement, échange avec d'autres agents, expérience).Les agents cognitifs possèdent des besoins, des envies, des croyances ou une éthique [88, 76]. Les agents cognitifs peuvent être autonomes, sont capables de se coordonner pour résoudre des conflits ou des problèmes les faisant se rapprocher d'individus vivant en société.

Les agents réactifs sont des agents ayant un comportement de réaction à des stimuli. Leur comportement de réaction a été inspiré par le comportement de certains insectes comme les fourmis [27] ou les Episyrphus balteatus [7]. Un agent réactif seul n'a pas un grand intérêt, mais un SMA contenant une multitude d'agents réactifs est capable de simuler un comportement de groupe intelligent. Cette approche considère qu'un système composé d'agents non-intelligents peut avoir un comportement intelligent. C'est par exemple le cas d'une fourmilière. Si l'on prend chaque entité de la fourmilière (une fourmi), indépendamment des autres, son comportement est extrêmement basique. Mais si l'on se réfère au comportement de la fourmilière, on peut remarquer qu'il s'agit d'une société organisée, hiérarchisée et fonctionnelle. D'après Van Dyke Parunak [64], il est impossible de prévoir de propriétés émergentes avec des systèmes réactifs du fait qu'ils n'ont pas de but ni de planification.

On constate que la séparation entre réactif et cognitif s'amenuise. Au départ, cette différence provenait en partie de la difficulté d'avoir un grand nombre d'agents avec des comportements complexes sur des ordinateurs avec une puissance de calcul limitée. Aujourd'hui, les machines sur lesquelles les SMA sont exécutés sont de plus en plus puissantes et on constate l'émergence de SMA avec des agents que l'on pourrait qualifier d'hybrides. Leurs comportements s'apparentent toujours à celui d'agents réactifs mais ils peuvent posséder une mémoire qui peut impacter leur réaction, ils sont capable de se coordonner et peuvent posséder des croyances.

Dans le but d'améliorer la modularité des SMA et la réutilisabilité des éléments produits dans ces SMA, Jean-Christophe Routier, Philippe Mathieu et Yann Secq proposent dans [68] d'offrir aux agents la capacité d'acquérir des compétences qui leur seraient données par d'autres agents. Ils partent du constat qu'un agent est une entité *capable d'agir* et que partant de ce postulat, il doit avoir les compétences pour réaliser cette action. Ils définissent une compétence comme *un ensemble de capacités* et décrivent un agent "atomique" comme un agent ne possédant que deux compétences de base : une pour interagir et une pour apprendre de nouvelles compétences. Ils expliquent qu'un agent possédant ces deux compétences sera capable d'évoluer, en acquérant de nouvelles compétences, afin d'améliorer ses capacités et de pouvoir changer son comportement

6. La jouabilité est le ressenti que le joueur a en jouant à un jeu vidéo. Est ce que les contrôles sont ergonomiques. Les règles du jeu sont elles cohérentes et procurent elles un plaisir à joueur.

voire même de changer le rôle qu'il a dans le système. Dans le domaine du jeu vidéo, l'utilisation de compétences pouvant être acquises est une approche très intéressante. Beaucoup de jeux se basent sur une évolution du personnage incarné par le joueur par l'acquisition de compétences le rendant plus fort, capable de nouvelles actions lui permettant d'avancer dans sa quête.

Nous visons le développement d'un système utilisant des agents hybrides. Ces agents auront un comportement de base proche d'un comportement d'agent réactif car ils ne feront que réagir à des interactions des autres agents. Cependant, ils seront aussi capables de réflexion, ils pourront posséder une mémoire et ils pourront aussi acquérir des compétences ce qui les fera se rapprocher des agents cognitifs. Notre système devra être assez permissif pour autoriser l'ajout de base de connaissance, de protocole de négociation ou de décision dans les comportements modélisés [70, 89].

2.2.2/ LES SYSTÈMES MULTI-AGENTS

L'agent prend son vrai sens lorsqu'il est immergé dans un système qui en regroupe d'autres. On parle alors de système multi-agents. Dans [32], Jacques Ferber propose une définition : "On appelle système multi-agents (ou SMA), un système composé des éléments suivants :

— un environnement $\mathtt{E}$, c'est-à-dire un espace disposant ou non d'une métrique,

— un ensemble d'objets $\mathtt{O}$. Ces objets sont situés, c'est-à-dire que, pour tout objet, il est possible, à un moment donné, d'associer une position dans $\mathtt{E}$. Ces objets sont passifs, c'est-à-dire qu'ils peuvent être perçus, créés, détruits et modifiés par les agents.

— un ensemble $\mathtt{A}$ d'agents, qui sont des objets particuliers ($\mathtt{A} \subseteq \mathtt{O}$), lesquels représentent les entités actives du système,

— un ensemble de relations $\mathtt{R}$ qui unissent des objets (et donc des agents) entre eux,

— un ensemble d'opérations $\mathtt{Op}$ permettant aux agents de $\mathtt{A}$ de percevoir, produire, consommer, transformer et manipuler des objets de $\mathtt{O}$,

— des opérateurs chargés de représenter l'application de ces opérations et la réaction du monde à cette tentative de modification, que l'on appellera les lois de l'univers."

On peut retenir de toutes ces définitions que les agents peuvent être perçus comme des êtres vivants autonomes dans la recherche de satisfaction de leur but, capables d'agir dans leur environnement, capables de communiquer, et qu'ils ont une représentation partielle de l'environnement dans lequel ils évoluent.

On peut aussi voir le SMA comme un ensemble organisé d'agents qui évoluent dans un même environnement et capables d'interagir entre eux. Le développement d'un système multi-agents se fait en intégrant donc ces quatre données identifiées dans l'approche voyelle AEIO définie : l'Agent, l'Environnement, les Interactions, et l'Organisation.

Il est tout de même à noter qu'il existe deux possibilités de développement de SMA : les SMA en temps discret et les SMA en temps réel.

2.2.3/ LA COMMUNICATION

Nous avons vu que la communication est un point central des SMA. Les agents sont capables de communiquer entre eux, mais aussi avec leur environnement. De nombreux travaux de recherche visent à améliorer le fonctionnement des communications dans les SMA.

Dans ses travaux, Yves Demazeau décrit l'approche MAGMA [26] permettant d'utiliser les SMA pour la résolution de problèmes complexes. Il décrit chacune des voyelles de l'approche AEIO en s'attardant sur les interactions. Une interaction entre deux agents est un échange d'informations et peut être considérée comme une communication. Il explique qu'au moment de la publication de ses travaux, les protocoles d'interactions entre agents étaient chacun développés pour une situation particulière. Il cite, entre autres :

- Un protocole de comportements hiérarchisés [30] où le comportement des agents est défini par 6 aspects (qui ? fait quoi ? quand ? où ? comment ? pourquoi ?) et où chaque niveau de la hiérarchie affine un aspect qui était général dans le rang précédent de la hiérarchie. Par exemple, un des aspects concerne la date à laquelle le comportement doit être exécuté (quand ?). Au rang N de la hiérarchie, cet aspect peut avoir la valeur "dans la semaine" et au rang N + 1, on peut retrouver deux valeurs différentes "aujourd'hui" et "demain ». Lors des interactions, chaque agent envoie le niveau le plus haut de sa hiérarchie. Si l'agent qui reçoit les informations détecte un conflit, il peut demander une version plus affinée des informations. Ce protocole permet des résolutions de problèmes entre agents, mais possède quelques limitations. Par exemple, dans sa forme première, il permet à un agent de fournir une information ou un service à un autre agent, mais ne permet pas à deux agents de réaliser une tâche ensemble si cette tâche nécessite une coordination entre eux.
- The Contract Net Protocol (CNP) [78] [71] [72] qui propose un protocole d'interactions par contrat. Dans ce protocole, chaque agent est capable d'exécuter certaines tâches et lorsqu'un agent a besoin d'exécuter une tâche qu'il ne maîtrise pas, il demande aux agents présents dans son réseau s'ils acceptent de l'aider. Un contrat est alors mis en place entre les deux agents. Ce système permet, contrairement au protocole de comportements hiérarchisés, aux agents de se coordonner. Smith explique même qu'il est capable de gérer une hiérarchie dans les tâches pour rendre l'exécution d'une tâche prioritaire par rapport à une autre. Cependant, un agent exécute une tâche jusqu'à ce qu'il ait fini, même si une requête plus prioritaire lui est envoyée et un agent n'est pas capable de refuser une tâche qui lui a été assignée. Certaines améliorations à Contract Net Protocol ont cherché à corriger ces défauts (ex.)[71]).

Afin de pallier au problème décrit par Yves Demazeau, certains travaux de recherche ont mis au point des protocoles de communication génériques. Cette généricité vise à permettre à des agents, ne faisant pas partie du même système, de communiquer. Les recherches ont mené au développement d'ACL (Agent Communication Language). Un ACL est un langage permettant de standardiser les échanges d'informations entre agents.

Lors d'un échange entre agents, il est important de séparer le protocole utilisé lors de l'échange (http, tcp, ...), du protocole de communication (KQML, FIPA ACL, ...), du contenu (propre à chaque système, mais pouvant nécessiter un vocabulaire commun ou ontologie). Deux des ACL les plus connus sont KQML et FIPA ACL :

— **KQML** (Knowledge Query and Manipulation Language) [59] est un ACL permettant à des agents de communiquer entre eux en se basant sur des primitives (appelées *performatifs*[7]) qui permettent aux agents de communiquer leurs requêtes à d'autres agents.

KQML se divise en trois calques : le calque de contenu permet, comme son nom l'indique de contenir le contenu du message à transmettre. Le calque de communication est le calque permettant la transmission du message. Le paramétrage de ce calque contient, entre autres, un identifiant unique pour le message ou l'identité de l'expéditeur. Et enfin le calque de message qui est le cœur de KQML. Ce calque est utilisé pour encoder le message dans le format KQML pour qu'il soit utilisable par d'autres agents utilisant le langage KQML.

Ce langage est le résultat de la coopération de quatre groupes de scientifiques : l'Advanced Research Projects Agency(ARPA), l'Air Force Office of Scientic Research (ASOFR), la Corporation for National Research Initiative (NRI) et la National Science Foundation (NSF). Cette association a mené à la mise au point de l'approche KSE (Knowledge Sharing Effort) visant à permettre l'échange de connaissances entre différents systèmes informatiques [60]. KQML permet de formater les données afin qu'elles puissent être "lues" par les agents fonctionnant avec KQML.

KQML étant un ACL, les informations transmises à un agent utilisant KQML peuvent être bien "lues", mais ne seront pas nécessairement comprises. Cette compréhension du message dépendra de la base de connaissances [77] de l'agent. Dans sa première version, KQML ne propose pas de sémantique, ce qui rend ce langage ambigu. Des évolutions ont été proposées pour améliorer KQML. Nous pouvons citer les travaux de Labrou [53] [36] [54] qui ont, entre autres, porté à 35 le nombre de *performatifs* et ajouté une description sémantique à ceux-ci pour faciliter la conversion entre les connaissances des agents et le contenu des messages. Mais comme le rappel Mahot [56], ces modifications présupposent "que les agents embarquent une base de connaissances, et la présentation des préconditions sous-entend également d'avoir des connaissances sur l'état interne de ses interlocuteurs, ce qui peut rapidement devenir une masse importante d'informations."

Et dans notre objectif d'optimisation, le fait d'avoir plus d'informations à traiter peut rapidement surcharger le système. Mais d'autres ACL ont été créées pour pallier les problèmes de KQML.

— **FIPA** (Foundation for Intelligent Physical Agent) est une fondation qui a pour but de définir des spécifications pour standardiser les interfaces agents [37] [62]. Ce standard définit, entre autres, des règles pour les interactions entre agents. La première version de FIPA (FIPA97) a défini trois axes de standardisation pour les agents (les communications, le management et l'intégration logiciel) complétés avec les interactions dans FIPA98. Les deux axes de standardisation qui nous intéressent le plus sont ceux de la communication et des interactions. Pour la communication, FIPA a mis au point un ACL, FIPA ACL, se basant sur le formalisme et la grande expressivité d'ARCOL[8] et la facilité d'utilisation de KQML. FIPA ACL est divisé en 5 niveaux :

- **niveau Protocol** permettant de définir les règles sociales entre les agents. Un acheteur demandant une information à un vendeur ;

7. Terme défini par Austin [8] désignant un verbe qui exécute une action lorsqu'il est énoncé. Dans la phrase : "Je vous déclare mari et femme", *déclare* est un performatif.

8. ACL développé par David Sadek en 1991 au sein de l'entreprise France Télécom [20]

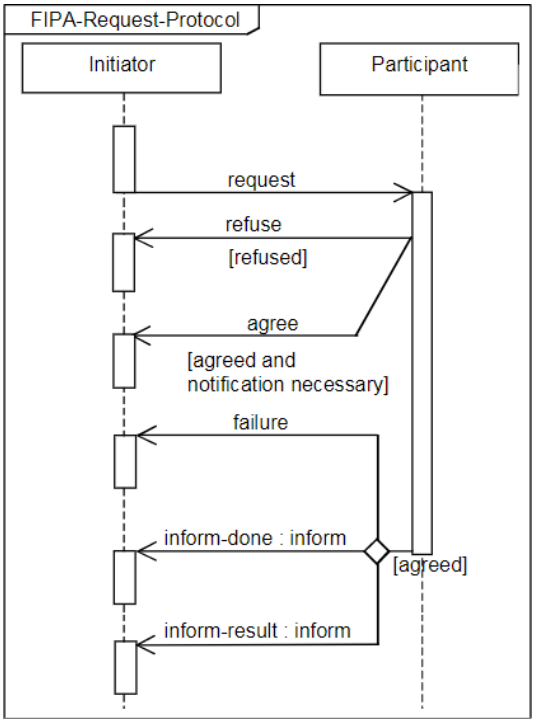

FIGURE 2.3 – Diagramme de description de FIPA-Request Protocol [4].

- **niveau Acte de communication** définissant la structure de la communication. Par exemple, une requête attendant une réponse ;
- **niveau Message** contenant les méta-informations des messages (Émetteur, récepteur, contexte, etc) ;
- **niveau Langage du contenu** définissant la grammaire et la sémantique associées pour permettre la lecture du message. Un exemple pourrait être PROLOG ;
- **Niveau Ontologie** permettant de définir le vocabulaire et la signification des termes utilisés dans le message.

Comme KQML, FIPA ACL fonctionne sur le principe de *performatifs*. Dans sa première version, FIPA ACL proposait 4 *performatifs* desquels les actes de langages étaient exprimés par composition. En 2000, FIPA ACL contenait 22 *performatifs*. Les messages de FIPA ACL contiennent les mêmes champs que les messages de KQML en plus de quelques nouveaux champs permettant, par exemple, de replacer un message dans un contexte (lui donner un lien avec d'autres messages précédemment envoyés).

FIPA a mis au point différents protocoles permettant de définir le comportement de certaines interactions afin que les agents puissent anticiper les réponses que leurs interlocuteurs pourront leur fournir. Par exemple, le *FIPA-Request protocol* est un protocole permettant de définir le déroulement d'une requête entre un agent initiateur et la réponse possible de l'agent participant (voir figure 2.3).

Que ce soit KQML ou FIPA, ces ACL permettent de définir un standard dans les échanges entre agents et sont réputés pour offrir une grande expressivité tout en n'alourdissant pas la taille des messages transmis entre les agents. Dans [59], Mckay et McEntire expliquent avoir compressé les messages pour obtenir un gain dans leur taille.

Cependant, la seule utilisation des ACL ne permet pas le fonctionnement complet d'interactions entre agents (qu'ils fassent partie d'un même système ou non). Nous l'avons déjà évoqué précédemment, il est nécessaire d'avoir un protocole qui se charge de l'envoi des messages, mais il est aussi nécessaire que les agents maîtrisent le vocabulaire avec lequel le contenu du message est créé afin de pouvoir comprendre le sens de ce message.

KQML et FIPA permettent aux agents de maîtriser chacun des vocabulaires en fonction des domaines d'application dans lesquels ils sont utilisés. Cela permet aux agents maîtrisant les mêmes vocabulaires de pouvoir communiquer et de se comprendre. Il est possible aussi que certains termes d'un vocabulaire existent dans un autre vocabulaire. Si l'on exclut les risques que cela inclut en termes de doubles sens d'un même terme, cela permet aux agents de pouvoir communiquer et se comprendre sans posséder le même vocabulaire. Par exemple, lorsqu'un agent astrologue veut connaître la position d'une planète il fait sa demande à un agent astronome. Ces deux agents n'ont pas le même domaine d'application, mais le terme "planète" est logiquement présent dans leurs vocabulaires respectifs.

Cependant, d'après Steels [79], ce fonctionnement utilisant des ontologies, des langages et des protocoles de communication définis à l'avance est mauvaise. Il cite quatre raisons à cela :

1. Il est difficile de déterminer tous les domaines d'application des applications multi-agents et ainsi difficile d'en déduire tous les vocabulaires, langages ou protocoles de communication.
2. Les systèmes multi-agents sont des systèmes ouverts ayant sans cesse de nouveaux besoins. Et il est impossible d'inclure ces nouveaux besoins dans des systèmes existants si tout est défini à l'avance.
3. Les systèmes multi-agents peuvent être distribués, sans contrôle central. Comment transmettre les mises à niveau des protocoles, langages ou vocabulaires aux agents ?
4. Dans le cas d'agents robotisés, l'ontologie est dépendante des capacités sensori-motrice de l'agent. Si l'agent robotisé évolue, son ontologie doit évoluer.

Mais toutes ces raisons n'ont pas réellement d'importance dans le domaine dans lequel nous travaillons. Les jeux vidéo ne sont pas ouverts dans le sens où les utilisateurs ne vont pas ajouter de nouveaux comportements aux agents ou intégrer de nouveaux agents avec un vocabulaire inconnu des autres agents.

Toujours dans ses travaux, Steels propose que les agents puissent apprendre à communiquer au contact des autres agents. Ainsi, un nouvel agent plus évolué (ayant subi une mise à niveau de son vocabulaire, de son langage ou de son protocole de communication) sera à même de faire évoluer les agents avec lesquels il communique. Ces agents capables d'apprendre et d'évoluer peuvent être considérés comme des agents cognitifs.

2.2.4/ L'ENVIRONNEMENT

Nous l'avons vu, Jacques Ferber définit l'agent comme ayant une connaissance partielle de son environnement et pouvant interagir avec lui. Mais dans le domaine des SMA, l'environnement est une notion très vague. Il peut avoir de multiples utilités et prendre de multiples formes.

Jacques Ferber [33] explique que l'environnement peut être représenté comme un système unique (centralisé) ou comme un système distribué. Dans nos recherches, nous utilisons des environnements centralisés. En effet, dans le domaine du jeux vidéo, même dans le cas de jeux multi-joueurs, il est préférable d'utiliser un serveur central pour réaliser les calculs de l'environnement afin d'éviter que l'un ou l'autre des joueurs ne puissent tricher.

Très souvent, l'environnement est utilisé comme un vecteur de communication en étant utilisé de plusieurs manières différentes. Nous dressons une liste non-exhaustive de ces possibilités :

— La méthode du tableau blanc consiste à laisser un message qui pourra être lu par tous les agents. Les messages laissés peuvent nécessiter certaines compétences pour être lus ou peuvent être limités à une zone géographique. Ces messages sont similaires aux phéromones laissés par les insectes dans leur environnement pour communiquer avec leurs semblables [27].

— La méthode du broadcast est le fait qu'un agent communique directement avec tous les autres agents présents dans un environnement sans que ces derniers n'aient à lire le message. Cette méthode est différente de la méthode du tableau blanc où les agents doivent interroger l'environnement pour lire les messages qu'il possède. Tout comme la méthode du tableau blanc, il est possible de filtrer les agents pouvant lire ces messages en imposant certaines compétences par exemple.

— La méthode pair-à-pair consiste en la mise en contact de deux agents afin qu'ils échangent entre eux sans passer par l'environnement.

Chacune de ces méthodes apporte un intérêt dans le domaine du jeu vidéo. La méthode du tableau blanc permet aux agents de voir les changements de l'environnement causés par la présence d'autres agents (ou de joueurs), des traces de pas au sol par exemple. La méthode broadcast permet, quant à elle, de simuler des communications de masse comme cela pourrait être le cas dans une communication par radio ou télépathie ou bien dans une discussion entre plusieurs agents. La communication pair-à-pair peut être utilisée afin de permettre aux agents de se notifier des interactions ne faisant pas partie des actes de langages (Par exemple pour notifier un contact entre deux agents).

Un environnement simple ne permet pas aux agents de communiquer en broadcast ou en pair à pair. Il manque en effet un moyen pour les agents de se mettre en relation. KQML et FIPA mettent en place des agents dits facilitateurs. Il s'agit d'agents particuliers permettant aux autres agents d'entrer en contact entre eux.

Les agents facilitateurs de KQML agissent comme des routeurs centralisant les informations sur les agents présents dans le système. Cette centralisation pose un problème de disponibilité du système du fait que si le facilitateur ne fonctionne plus, les agents seront incapables de se contacter.

FIPA définit deux types d'agents facilitateurs :

1. **Agent Management System (AMS)** : permet aux agents de récupérer une liste des agents avec qui ils peuvent communiquer. Cet agent agit un peu comme le service des pages blanches d'un annuaire. Il est aussi en charge de la création et la destruction des agents du système qu'il manage.
2. **Directory Facilitator (DF)** : permet aux agents de récupérer une liste des services que chacun des autres agents met à disposition. Cet agent fonctionne comme le service pages jaunes d'un annuaire.

Un des objectifs de l'AMS de FIPA est d'améliorer la stabilité du système. Lorsque les travaux sur les facilitateurs ont été menés, c'était pour pallier le fait que les systèmes de découvertes de services (qu'ils soient agents facilitateurs ou non) fonctionnaient principalement de manière centralisée.

Chaque agent qui propose des services va envoyer la liste de ceux-ci à l'agent DF, au moment où il entre dans le système. Lorsqu'un agent A a besoin d'un service, il demande à l'agent DF de le mettre en liaison avec un agent pouvant exécuter ce service.

Nous l'avons vu précédemment, les moteurs physiques permettent, entre autres, de connaître le positionnement de chaque agent. Dans les SMA, l'environnement peut avoir ce rôle de renseigner les agents sur le positionnement des autres agents. Le positionnement des agents et la capacité de chaque agent de pouvoir connaître la position des autres agents ont une grande importance dans les SMA. Ainsi, Philippe Mathieu [57] propose 4 *patterns* fondamentaux pour le positionnement des agents :

— Le premier *pattern* permet de **lister** le voisinage d'un agent en parcourant tous les agents du système pour chercher ceux qui lui sont proches.

— Le deuxième *pattern* **divise** l'environnement en grille (environnement discret) et liste les cellules proches de celle où se trouve l'agent pour déterminer son voisinage. Dans ce pattern, les agents d'une même cellule sont considérés comme des voisins proches et les agents présents dans deux cellules considérées comme proches sont eux-mêmes considérés comme proches les uns des autres. Ce deuxième *pattern* donne aussi un algorithme pour le déplacement des agents dans cet environnement.

— Le troisième *pattern* réalise la même chose que le deuxième *pattern*, mais en **transcrivant** un environnement en espace continu vers un environnement discret afin de pouvoir appliquer les règles du deuxième *pattern*.

— Le quatrième *pattern* permet aux agents de **connaître** leur positionnement en fonction de la proximité *social* qu'ils ont avec les autres agents.

Dans le but de formaliser les interactions, Yoann Kubera, Philippe Mathieu et Sébastien Picault ont développé *Interaction-Oriented Design of Agent simulations* (*IODA*), une approche orientée interactions visant à formaliser les interactions et permettant de les définir par le biais d'une matrice. Les interactions de *IODA* représentent chaque action pouvant être réalisée par les agents. Chaque interaction définie un nombre d'agents impactés par elle et comment et sous quelles conditions ces agents sont impactés. Pour ce faire, les interactions possèdent des conditions qui sont la conjonction :

— De pré-conditions permettant de définir sous quelles conditions l'interaction peut être initiée. Par exemple, dans le cas d'une interaction où l'agent se nourrit, une pré-condition serait *la nourriture n'est pas avariée* ;

— De déclencheurs permettant d'indiquer quels événements déclenchent une interaction. Toujours dans le cas d'une interaction où l'agent se nourrit, le déclencheur serait *l'agent à faim*.

Dans IODA, une interaction décrit aussi les agents sources (ceux qui initient l'interaction) et les agents cibles (ceux qui la subissent) ainsi que la cardinalité de cette interaction sous la forme $(card_S(I), card_T(I))$ avec $card_S(I)$ représentant le nombre d'agents sources et $card_T(I)$ représentant le nombre d'agents cibles.

Enfin, une interaction dans *IODA* possède une liste d'actions correspondant à des primitives qui seront exécutées par les agents qui exécutent l'interaction. Les primitives sont

Source \ Target	∅	Grass	Sheep	Goat	Wolf
Grass	(Grow)				
Sheep	(Move)	(Eat, d = 0)	(Procreate, d = 1)		
Goat	(Move)	(Eat, d = 0)		(Procreate, d = 1)	
Wolf	(Move)		(Eat, d = 3)	(Eat, d = 3)	(Procreate, d = 1)

FIGURE 2.4 – Matrice d'interaction de *IODA*.

similaires à des méthodes de la programmation orientée objets. Elles permettent de définir les actions réalisables par les agents.

La modélisation des interactions que les agents peuvent exécuter se fait par le biais d'une matrice comme celle présente dans la figure 2.4. Dans cette matrice, les agents sources sont dans la colonne de gauche et les agents cibles dans la ligne supérieure. A l'intersection d'une ligne et d'une colonne on peut trouver les interaction qu'un agent source peut exécuter à destination d'un agent cible. La valeur *d* correspond à la portée de l'interaction.

IODA permet aussi de définir des perceptions. Dans *IODA*, chaque agent peut percevoir les agents dans leur voisinage (sous entendu dans leur *halo*). Ce concept de *halo* peut cependant avoir un inconvénient qui est qu'un agent ne possède qu'une seule manière de percevoir car il ne possède qu'un seul *halo*. Il est alors complexe, mais non impossible, de développer une perception orale et visuelle pour un même agent par exemple.

Dans nos travaux de recherche, nous avons décidé de faire de l'environnement un vecteur de communication qui a aussi comme rôle celui de facilitateur. Nous nous basons sur les agents facilitateurs définis par FIPA. Nous modifions cependant l'agent DF pour qu'il puisse filtrer les réponses qu'il donne aux requêtes des agents. Cela permet de générer un comportement où un agent propose certains services à une liste réduite d'agent, en fonction de leurs liens sociaux, leur hiérarchie ou autre. L'environnement aura aussi pour rôle de faire le lien entre le moteur physique du système et les agents afin que ces derniers puissent obtenir des renseignements sur le positionnement des autres agents du système.

2.3/ SYSTÈMES MULTI-AGENTS ET GESTION DU TEMPS

Nous avons décrit précédemment que les SMA peuvent être développés pour fonctionner en temps réel ou non. Le choix sera souvent dicté par les besoins du système : Y a nécessité d'observer les résultats du système pendant qu'il s'exécute, ou est-il possible d'analyser ces résultats lorsque le système aura terminé son exécution ?

Les systèmes en temps réel sont nécessaires lorsque le système doit répondre à des informations provenant de périphériques externes. Il peut s'agir de capteurs dans des ateliers, des usines ou dans les villes [45] [50] [91]. Dans ces exemples, le système doit réagir en temps réel afin d'adapter son comportement aux nouvelles données qu'il a reçu des périphériques externes. La littérature définit aussi un système temps réel comme un système qui n'est pas seulement caractérisé par ses fonctionnalités (temporelles) mais aussi et surtout par des contraintes de temps [66] [74]. Plus spécifiquement, il s'agit de systèmes dans lesquels les tâches exécutées possèdent une *deadline*.

Dans le domaine qui est le nôtre, lorsque l'utilisateur souhaite réaliser une action, il utilise un périphérique pour signaler au système l'action qu'il souhaite réaliser. C'est pourquoi

les SMA que nous souhaitons développer à l'aide de nos travaux doivent être exécutés en temps réel. De plus les animations des agents, la physique des projectiles ou les effets météorologiques ont besoin d'un rendu fluide pour que l'expérience de l'utilisateur soit optimale.

Dans cette section nous abordons les recherches menées afin de développer des SMA en temps réel. Nous voyons que l'optimisation de la consommation des ressources n'est qu'une partie de l'état de l'art des systèmes temps réel et que la fréquence de mise à jour du système n'est pas forcément garantie.

Nous commençons par donner quelques précisions sur les termes employés dans cette section. Nous continuons avec l'état de l'art des recherches visant au développement de SMA en temps réel en abordant les différents axes de recherches envisagés. Nous concluons enfin sur les questions de gestion du temps dans les SMA.

2.3.1/ PRÉAMBULE

Avant de continuer cette section, nous allons faire un rapide rappel en ce qui concerne le temps dans les systèmes informatiques. Il est important pour pouvoir bien aborder la suite de cette section de comprendre la différence entre temps réel, temps discret et temps continu. Le terme continu que nous utilisons est celui de l'aspect des théories mathématiques décrit dans [61] par Anne Nicolle.

Dans cette définition du temps continu, il est mis en opposition au temps discret. Un système à temps discret est un système dont le temps "avance" par pas. Dans certains systèmes chaque *pas* représente la même durée (Gama [81] [42] [6], NetLogo [82] [87]) quand, dans d'autres, chaque *pas* est exécuté lorsque le système en a besoin (synchronisation dans des systèmes distribués) et sa durée peut-être différente des autres *pas*. Un système en temps continu est par opposition un système dont l'exécution n'est pas découpée en *pas*.

Shine Engine est un système avec un temps discret. Chaque boucle du système représente un *pas*, d'une durée pouvant être différente de celle des autres *pas*. En informatique, il n'existe pas réellement de temps réel. Dans le domaine du jeu vidéo, le terme temps réel a une autre signification. Il s'agit d'une composante du gameplay. Un jeu en temps réel est un jeu dans lequel il n'y a pas de temps d'arrêt, le temps dans le jeu se déroule à vitesse constante, sans interruption. Il est à mettre en opposition aux jeux au tour par tour dans lesquels les joueurs ont, entre chacune de leurs actions, un temps de réflexion.

Dans la suite de ce mémoire, le terme temps réel est toutefois assimilé au temps réel informatique. Malgré le fait que Shine Engine ne soit pas en temps réel, nous estimons nous rapprocher du temps réel lorsque l'affichage du jeu est fluide. Cette fluidité est atteinte à 30 images par seconde. La suite de cette section explique plus en détails les notions de temps dans les SMA.

2.3.2/ SYSTÈMES MULTI-AGENTS EN TEMPS RÉEL

La notion de temps réel est indépendante des notions de temps discret et continu. Elle permet à des signaux extérieurs au système, comme les informations provenant des

contrôleurs de jeu (manettes, souris, clavier, écran tactile, ...), d'avoir un impact sur son fonctionnement.

Cette définition est parfaitement adaptée aux besoins du domaine des jeux vidéo. En effet, lors de son exécution, un jeu vidéo affiche un certain nombre de frames par seconde et, pour que le rendu soit fluide et ne gâche pas l'expérience de l'utilisateur, il est nécessaire d'en afficher un maximum.

Rappelons qu'entre chaque frame le jeu doit réaliser des calculs afin de faire évoluer les informations qu'il possède : position de chaque entité, calculs physiques ou de lumières, etc. Tous ces calculs doivent être réalisés avant l'affichage de l'image suivante, soit dans un temps maximum de 33ms (pour un rendu de 30 images par seconde). Le système que nous cherchons à mettre en place doit assurer que tous les agents se mettent à jour toutes les 33ms maximum, sachant qu'il faut aussi du temps pour les calculs physiques, de lumières, d'ombrages et pour exécuter les animations et autres sons.

Les recherches concernant le temps réel dans les SMA et que nous avons étudiées cherchent principalement à optimiser les systèmes afin qu'ils répondent le plus rapidement possible [84, 49].

Des simulateurs comme Gama ou NetLogo ont un fonctionnement en temps réel. Il est, en effet, possible d'observer les résultats d'une simulation pendant qu'elle est exécutée. Il est aussi possible de modifier des paramètres de cette simulation pendant qu'elle est exécutée.

Pour répondre à notre besoin d'assurer que tous les agents se mettent à jour en moins de 30ms, nous avons cherché du côté des planificateurs de tâches [12, 24]. Un planificateur est un outil permettant de déterminer l'ordre d'exécution des tâches d'un système afin d'optimiser le temps d'exécution complet du système. Les planificateurs prennent en compte les priorités des tâches et peuvent mettre en pause certaines tâches afin d'en exécuter d'autres plus prioritaires. Ils sont aussi capables de gérer les dépendances entre les tâches (par exemple, lorsqu'une tâche a besoin du résultat d'une autre tâche).

Enfin, et c'est le point qui nous intéresse le plus, certains planificateurs sont capables de donner des *deadlines* aux tâches et de les arrêter si elles prennent plus que le temps qui leur a été assigné.

Dans les SMA, un planificateur de tâches aura pour rôle de déterminer l'ordre d'exécution des tâches de chaque agent en fonction des priorités et des deadlines de chacune des tâches. Les planificateurs de tâches peuvent aussi avoir pour but de déterminer des *deadlines* pour les tâches que les agents exécutant devront respecter.

Dans notre système, la *deadline* devrait être fixée au temps maximum que nous souhaitons pour l'exécution de la frame. Si cette *deadline* n'est pas respectée, les tâches en cours d'exécution devraient être mise en pause et redémarrées au début de la frame suivante. Les planificateurs que nous avons étudiés ne permettent pas de tels fonctionnements. De plus, un tel planificateur aurait un coup en temps de calcul et pourrait impacter le temps de calcul des agents.

Le concept de *deadline* est aussi utilisé par les *systèmes à événements discrets* comme *DEVS* [90, 35]. Dans DEVS chaque système se trouve dans un état qui fixe son comportement. Le système peut recevoir des activations qui le feront changer d'état. Chaque état dans lequel le système peut se trouver possède une *deadline* qui, si elle est atteinte, aura pour conséquence de faire changer d'état le système. Ce principe est intéressant

pour éviter que des systèmes prennent trop de temps sur un état en particulier. Cependant, dans notre contexte, l'objectif n'est pas de partager le temps de calcul entre tous les agents mais que tous les agents terminent leur calcul en un temps réduit.

D'autres travaux de recherche ont été menés afin d'approcher une exécution en temps réel des SMA notamment en travaillant sur l'architecture matérielle des systèmes. Cette architecture matérielle a une grande importance dans l'efficacité du système. Une architecture multi-coeur sera plus efficace qu'une architecture mono-coeur (à caractéristiques identiques) tout comme une mémoire sera plus efficace qu'une autre si ses vitesses de lecture et d'écriture sont plus élevées que celles d'une autre mémoire. Des recherches ont aussi été menées afin d'optimiser la consommation de ressource, notamment de ressources processeur [75] [9] [10].

Paul Richmond et Daniela Romano dans [67] utilisent des optimisations sur GPU afin d'utiliser la puissance de calcul des processeurs graphiques. Mais Emmanuel Hermellin, Fabien Michel et Jacques Ferber [46] explique que cette optimisation est efficace pour des SMA à base d'agents réactifs et qu'il est nécessaire d'utiliser des système hybrides (avec calculs sur CPU et GPU) en ce qui concerne des agents plus évolués comme les agents cognitifs. Ils expliquent qu'il existe des systèmes hybrides permettant les calculs sur CPU et GPU permettant une plus grande optimisation que ce que proposent les systèmes 100% sur GPU. Cependant, les ressources GPU sont, dans les jeux vidéo, extrêmement précieuses et il n'est pas envisageable de consommer de la puissance GPU pour d'autres calculs que ceux de rendu.

Nous avons déjà abordé les communications entre les agents. Du côté des communications Julian et al. ont développé RT-MESSAGE [49], un modèle basé sur MESSAGE [23]. RT-MESSAGE est développé pour parer l'impossibilité pour MESSAGE de fonctionner dans des systèmes en temps réel.

2.4/ MODÉLISATION

Les équipes de développement de jeux vidéos sont composées d'acteurs multiples et variés. On y trouve des développeurs en informatique, des game designers, des graphistes, des animateurs, des *level designers*, etc. Chaque corps de métier a un rôle dans le développement d'un jeu et utilise des outils qui sont propres à ses besoins. Cependant, certains de ces acteurs ont parfois besoin de faire du développement de code informatique afin de réaliser leurs travaux. Il s'agit par exemple d'un game designer qui aurait besoin de définir le comportement d'un personnage d'un jeu ou un level designer qui aurait besoin de créer un script et l'intégrer dans un niveau du jeu.

Le travail en binôme est alors obligatoire afin que le game designer ou le level designer se fasse aider par un développeur pour créer ce dont ils ont besoin. Nous avons alors réalisé des recherches dans le but de développer un système permettant aux non-développeurs de pouvoir participer à la création de jeux sans avoir recours à un développeur. Que ce soit dans une équipe dans un studio ou en tant que créateurs indépendants.

Dans cette section, nous nous attardons sur ce qui nous a semblé être la meilleure solution pour répondre au besoin de simplicité : la création par modélisation de graphes de nœuds. Nous voyons quels logiciels utilisent déjà cette solution, quelles pourraient être les limites de ces systèmes et quels en sont les avantages. Nous commençons par la

présentation d'outils de modélisation graphique développés afin d'assister les chercheurs dans le développement de système multi-agents et nous continuons par des logiciels permettant le développement de jeux vidéo en minimisant l'utilisation de code.

2.4.1/ DANS LE MONDE ACADÉMIQUE

NetLogo [82] et GAMA [81] [2] sont deux outils existants qui offrent simplicité d'utilisation et liberté de modélisation.

En effet, ces deux outils offrent chacun un langage simplifié afin que l'utilisateur puisse facilement modéliser des SMA. Cependant, l'apprentissage d'un langage, aussi simple soit-il, peut représenter un frein pour certains utilisateurs. Par exemple, les figures 2.5 et 2.6 exposent le code et le résultat obtenu afin d'afficher un simple cercle tournant. Malgré la simplicité apparente de l'utilisation de NetLogo, le code afin de créer ce cercle est trop complexe pour un utilisateur novice.

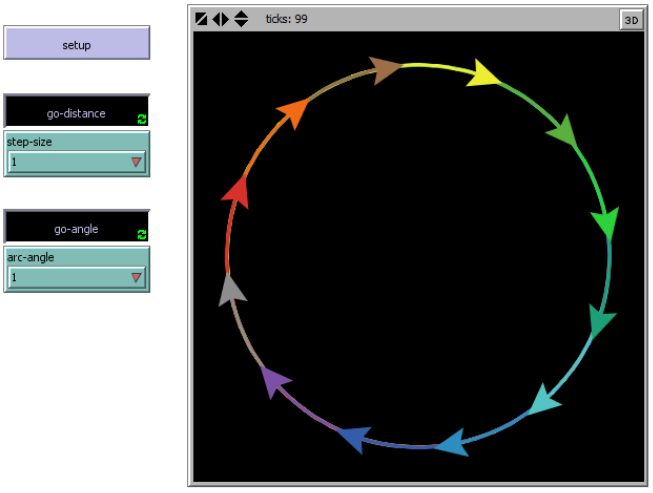

FIGURE 2.5 – Cercle tournant sur lui même développé avec NetLogo et le code de la figure 2.6

Il faut tout de même noter que pour accompagner les utilisateurs, un grand nombre d'exemples prêts à l'exécution sont fournis lors de l'installation de ces deux outils. Bien qu'ils soient simples d'utilisation, ils permettent tout de même de modéliser un grand nombre de comportements d'autant plus que nous avons la possibilité d'ajouter des plugins ou des extensions. Ces ajouts permettent de modifier le comportement natif des outils en y ajoutant des fonctionnalités spécifiques. Il est ainsi possible d'ajouter des plugins permettant, par exemple, la modélisation entièrement graphique d'un SMA dans GAMA (GAMAGraM [80]) ou la possibilité de charger des données GIS dans NetLogo [1].

Chacune des plateformes que nous avons citées laisse une certaine liberté quant aux choix de modélisation, mais elles restent assez limitées dans les possibilités offertes. Par exemple, GAMA ou MAGéo [3] [55] sont spécialisés dans les simulations spatiales, ce qui induit une restriction sur le type de simulation modélisable.

Ces outils scientifiques ne sont cependant pas les seuls à proposer une modélisation simplifiée de SMA ou de systèmes similaires.

```
globals [radius]

to setup
  clear-all
  set radius max-pxcor - 2
  ;; make 12 turtles, equally spaced in heading
  create-ordered-turtles 12 [
    ;; move to edge of circle
    fd radius
    ;; turn to face clockwise
    rt 90
    ;; bigger turtles are easier to see
    set size 3
    ;; thicker line is easier to see
    set pen-size 3
    ;; leave a trail
    pen-down
  ]
  reset-ticks
end

to go-distance
  ask turtles [ arc-forward-by-dist step-size ]
  tick
end

to go-angle
  ask turtles [ arc-forward-by-angle arc-angle ]
  tick
end

;; This is the core procedure.  It moves the turtle to the next point
;; on the circle, the given distance along the curve.

to arc-forward-by-dist [dist]  ;; turtle procedure
  ;; calculate how much of an angle we'll be turning through
  ;; (essentially converting radians to degrees)
  let theta dist * 180 / (pi * radius)
  ;; turn to face the next point we're going to
  rt theta / 2
  ;; go there
  fd dist
  ;; turn to face tangent to the circle
  rt theta / 2
end

to arc-forward-by-angle [angle]
  ;; turn to face the next point we're going to
  rt angle / 2
  ;; calculate the distance we'll have to move forward
  ;; in order to stay on the circle. Go there.
  fd 2 * radius * sin (angle / 2)
  ;; turn to face tangent to the circle
  rt angle / 2
end
```

FIGURE 2.6 – Code développé dans NetLogo afin d'obtenir le cercle de la figure 2.5

2.4.2/ DANS LE MONDE DU JEU VIDÉO

Nous l'avons vu quand nous avons abordé les moteurs de jeux vidéo, de nombreux éditeurs existent afin d'aider au développement de jeux. Ces éditeurs permettent de créer les niveaux des jeux vidéo et certains vont encore plus loin en proposant des outils pour le développement des comportements des entités faisant partie de ces jeux. Même s'ils ne sont pas tous basés sur le paradigme multi-agents, il est tout de même intéressant de s'attarder sur ces outils puissants.

Pour ce qui est de la facilité d'utilisation, Game Maker est certainement l'un des mieux classés. Il permet de développer des jeux en 2D avec une simplicité presque enfantine. Avec GameMaker, il est possible de créer des objets et d'appliquer à chaque objet des événements, une physique, des variables, des collisions, des texture pour son rendu, etc. Le tout se fait par le biais d'une interface graphique épurée et complète (cf Fig. 2.7). Et pour les utilisateurs plus accomplis, un système de script basé sur Delphi vient compléter l'interface graphique pour développer des comportements plus complexes.

Dans le domaine du jeu vidéo et des moteurs de jeu, Unreal Engine possède une place prédominante. Utilisé dans un grand nombre de productions, il possède lui aussi un éditeur de comportements graphiques appelé BluePrint. BluePrint permet de créer des comportements assignables aux entités du système. Ces comportements sont modélisables par le biais d'un système de graphe de nœuds où chaque nœud possède un rôle (cf figure 2.8). On retrouve parmi tous les nœuds, des nœuds variables, des nœuds fonctions, des nœuds événements, des nœuds conditions, etc. Tout comme pour GameMaker, Unreal Engine permet aux développeurs accomplis de développer des parties de leurs jeux en se servant d'un langage de programmation textuel. Mais la création d'un jeu en utilisant uniquement Blueprint est possible.

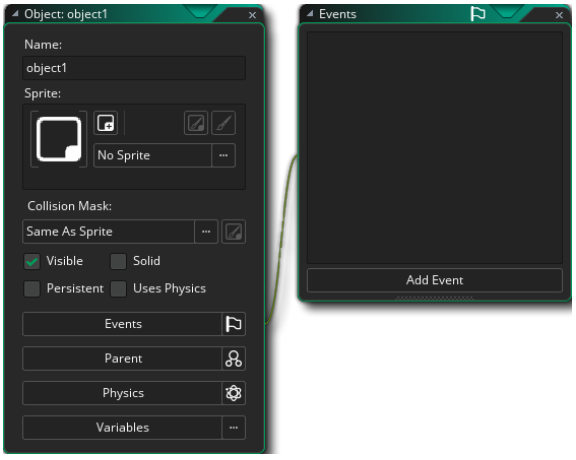

FIGURE 2.7 – Interface graphique de paramétrage d'un objet GameMaker

Malgré son approche proposant une interface simple à prendre en main dû à son utilisation d'un langage graphique plutôt que d'un langage textuel, Unreal Engine reste compliqué à prendre en main. Ceci résulte de l'obligation de maîtriser la logique de programmation. En effet, même graphiquement, appréhender le fonctionnement des boucles, des conditions et des variables peut être un frein au développement d'un jeu.

2.5/ SYNTHÈSE

Nous dressons ici une analyse de l'état de l'art et en extrayons les objectifs qui ont été les nôtres tout au long de nos recherches.

2.5.1/ ANALYSES

La première information que nous extrayons de cet état de l'art est qu'il est tout à fait possible de développer un jeu vidéo en utilisant le paradigme multi-agents. De nombreux jeux vidéo sont développés à base de SMA et, même s'il s'agit principalement de *serious game*, il n'y a aucun problème à le faire avec des jeux à but uniquement ludique et artistique.

Nous avons abordé dans l'état de l'art, les travaux réalisés par Philippe Mathieu [57] qui proposait 4 patterns fondamentaux pour le positionnement des agents, ainsi que les travaux de Yoan Kubera [52] qui proposait que chaque action que les agents réalisent soit des interactions. Même si ces recherches sont pertinentes, il ne nous semble pas judicieux de les intégrer à SAMP. En effet, SAMP est un système utilisé pour le jeu vidéo et à ce titre il va utiliser un moteur physique. Les SMA que nous avons étudiés jusqu'à maintenant ne sont pas assez ouverts pour l'intégration de ce genre d'outils. Dans IODA [52], l'agent interagit directement avec son environnement pour lui indiquer qu'il se déplace ce qui représenterait un dédoublement d'informations dans le cas d'une utilisation couplée à un moteur physique. Et des travaux comme ceux de Philippe Mathieu [57] présentent des algorithmes permettant de ne réaliser qu'une partie des tâches que réalisent les moteurs physiques (le positionnement) en ne proposant pas tout ce qu'un moteur physique propose : simulation des forces de poussées ou de frottements, calcul des collisions, gravité,

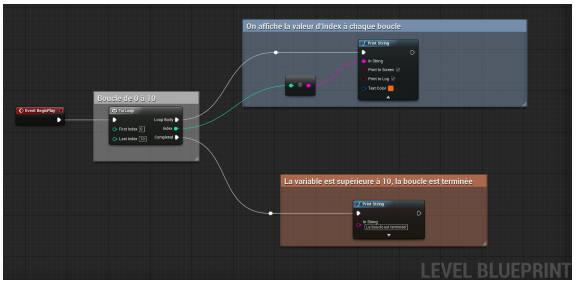

FIGURE 2.8 – Exemple d'un Blueprint développé à l'aide d'Unreal Engine.

etc.

Les outils existants permettant la modélisation de SMA ne permettent pas d'intégrer facilement un moteur physique à leur fonctionnement de base. C'est pourquoi nous avons décidé de développer un nouveau modèle de SMA qui utiliserait nativement un moteur physique et capable d'utiliser n'importe quel moteur physique existant.

Dans leur objectif initial, les facilitateurs permettent aux agents d'exposer quels services ils mettent à disposition des autres agents. Ce fonctionnement est très intéressant dans le cas où chaque agent collabore avec les autres. Dans le cas d'un jeu vidéo, les agents peuvent avoir le choix de communiquer ou non les capacités qu'ils possèdent mais ils peuvent aussi mentir à propos de ce qu'ils sont capables ou non de faire.

Aidés par l'état de l'art, nous pensons que limiter au maximum le besoin d'utiliser un langage de programmation textuel est une bonne solution pour faciliter l'utilisation de notre contribution. En se basant sur des outils comme le Blueprint d'Unreal Engine, il nous semble pertinent de chercher à développer un système permettant de modéliser des comportements à l'aide de graphe de nœuds. Cependant, la complexité d'utilisation de Blueprint nous a amené à réfléchir à proposer plusieurs niveaux d'utilisation de SAMP. Il est nécessaire de cibler les besoins et compétences de chaque utilisateur pour leur offrir différentes approches de SAMP : la possibilité de développer des comportements primaires ou de ne développer que des comportements abstraits.

2.5.2/ OBJECTIFS

Forts de cette analyse, nous nous sommes fixé 3 objectifs :

— développer un logiciel permettant de modéliser, générer et exécuter des systèmes multi-agents sans l'utilisation de langage textuel : utilisation d'une interface graphique pour le paramétrage des SMA, modélisation graphique, etc ;

— intégrer dans un même système les spécificités des SMA et des jeux vidéos : agents, communications, moteur physique, boucle de jeu, etc ;

— que les systèmes développés à l'aide de ce logiciel fonctionnent en temps réel.

Dans la suite de ce mémoire, nous décrivons comment nous avons mis au point chaque partie de SAMP et comment chaque partie interagit avec les autres.

II

TRAVAUX RÉALISÉS

3

EXEMPLE FIL ROUGE

Sommaire

Nous définissons dans ce chapitre un exemple qui servira de fil conducteur pour le reste de ce mémoire. Cet exemple est accompagné de sa modélisation dans SAMP afin de pouvoir appuyer nos explications. Nous avons décidé de mettre en place un système de proies-prédateurs car il est très répandu dans la littérature multi-agents et qu'il peut s'adapter facilement au domaine des jeux vidéo.

3.1/ DÉFINITION DE L'EXEMPLE

L'exemple choisi est un système proies-prédateurs. Il n'existe pas de standard pour les systèmes de proies-prédateurs, mais de nombreux travaux de recherche ont utilisé cet exemple [29, 17, 16].

Dans notre exemple, il y a 3 types d'agents différents : les loups, les moutons et l'herbe. Les loups peuvent manger les moutons et se reproduire. Les moutons peuvent manger l'herbe et se reproduire. L'herbe peut s'étendre afin de recouvrir des zones non-herbeuses.

Dans cet exemple, les loups et les moutons possèdent une propriété représentant leur niveau de faim. Celle-ci diminue au fil du temps pour indiquer leur réserve et déclencher la faim. Les moutons et les loups se déplacent aléatoirement jusqu'à ce que leur niveau de faim atteigne ce seuil. Lorsqu'il est atteint, ils se mettent à la recherche de nourriture. Lorsque cette nourriture est trouvée, ils se rapprochent d'elle, la mangent et recommencent à se déplacer aléatoirement. Si un mouton ou un loup ne trouve pas de nourriture et que son niveau de faim atteint 0, il meurt.

Lorsqu'un loup ou un mouton a fini de manger, il peut entrer dans un état de recherche d'un partenaire pour se reproduire. Il entre alors dans la même phase de recherche que pour la nourriture avec une cible différente. Lorsqu'il trouve un partenaire, il se rapproche de lui et ils se reproduisent si le partenaire cherche aussi à se reproduire. Il recommence ensuite à se déplacer aléatoirement. S'il ne trouve aucun partenaire et que son niveau de faim le requiert, il recherchera de la nourriture avant de recommencer à chercher un partenaire.

Lorsqu'un agent herbe est en contact avec un emplacement de l'environnement qui n'est pas encore colonisé par de l'herbe, il est capable de le coloniser. La colonisation consiste en la création d'un nouvel agent sur l'emplacement jusque-là vierge. Lorsqu'un nouvel agent herbe apparaît sur un emplacement de l'environnement, il met un certain temps à devenir comestible. Lorsqu'un agent herbe est attaqué (mangé) par un mouton, il meurt et l'emplacement sur lequel il se trouvait redevient vierge. Pour plus de facilité, nous avons découpé l'environnement en grille et chaque cellule de cette grille est un emplacement pouvant être colonisé par de l'herbe.

Dans la suite de cette section, nous présentons comment la population est modélisée. Nous donnons pour chaque type d'agent, son comportement, ses propriétés et ses compétences.

3.2/ MODÉLISATION DE LA POPULATION

L'exemple décrit met en place plusieurs agents ayant chacun un comportement défini. Ces agents sont la population de la simulation, et SAMP permet de définir cette population : type d'agent présent, propriété des agents de chaque type, nombre d'agents pour chaque type, capacités.

Dans cette section, nous allons exposer la modélisation de la population de notre exemple ainsi que la modélisation des instanciations des agents.

TABLE 3.1 – Type d'agent de la simulation

Nom	Héritages	Cpt	Propriétés	Compétences
Etre vivant	—	—	Points de Vie	
Mouton	Etre vivant	Cpt Mouton	Résistance Niveau de faim Seuil de faim Perte faim à l'arrêt Perte faim en déplacement	Être attaqué Être vu Voir Se reproduire Attaquer
Loup	Etre vivant	Cpt Loup	Force Niveau de faim Seuil de faim Perte faim à l'arrêt Perte faim en déplacement	Être vu Voir Se reproduire Attaquer
Herbe	Etre vivant	Cpt Herbe	Chance de colonisation	Être attaqué Être vu Coloniser

La table 3.1 montre comment la population de notre système est modélisée. Nous voyons que la population de notre exemple comporte 4 types d'agents : être vivant, mouton, loup et herbe. Les agents mouton, loup et herbe héritent du type être vivant.Nous aurions pu factoriser les propriétés communes des *loups* et des *moutons* en les faisant hériter d'un type *animal* qui aurait possédé les propriétés *niveau de faim*, *seuil de faim*, *perte de faim*. Nous constatons que les moutons suivent le comportement *mouton*, les loups suivent le comportement *loup* et l'herbe suit le comportement *herbe*. Tous les êtres vivant possèdent des points de vie. Nous avons donné une résistance aux agents de type mouton

et une force aux agents de type loup ainsi que trois propriétés liées au fonctionnement de la faim :

— Niveau de faim : correspond au niveau de faim lorsque la barre est totalement remplie. On estime que lorsqu'un agent mange, sa barre se remplit entièrement ;

— Seuil de faim : correspond au niveau de faim à partir duquel l'agent commencera à chercher à se nourrir ;

— Perte de faim par seconde : correspond à la perte du niveau de faim d'un agent à chaque seconde. Ce niveau de faim évolue en fonction de l'état dans lequel les agents se trouvent.

Les agents de type herbe possèdent une propriété indiquant, pour chaque seconde, le pourcentage de chance de coloniser un emplacement de l'environnement adjacent. Le test est effectué, pour chaque emplacement vierge à proximité de chaque agent herbe. Si un emplacement vierge est voisin de deux agents *herbe*, le test sera effectué pour chacun des deux agents.

4

SHINE AGENT MODELING PLATFORM

Sommaire

Dans ce chapitre, nous répondons aux besoins spécifiques évoqués dans la synthèse de l'état de l'art. En effet, ces derniers nous ont amenés à développer une nouvelle approche des SMA : Shine Agent Modeling Platform (SAMP). SAMP est une approche composée de trois modules :

— SAMP-M qui est le méta-modèle des SMA générés par SAMP ;

— SAMP-E qui est un outil graphique d'édition de SMA ;

— SAMP-X qui est un outil de génération de code. SAMP-X génère le code des SMA modélisés à l'aide de SAMP-E et respectant les règles définies par SAMP-M.

Pour ce faire, nous présentons Shine Agent Modeling Platform en décrivant chacun des trois modules qui le composent. Nous commençons par décrire les fondements de SAMP, les principes de bases que nous avons décidé de mettre en place afin de développer notre approche. Nous continuons en décrivant le fonctionnement des interactions, des compétences et des agents et les règles qui régissent leurs liens dans SAMP-M. Nous expliquons comment chacun de ces éléments est paramétrable dans SAMP-E et quels impacts ces paramètres auront sur le système final généré par SAMP-X.

Nous continuons en décrivant le fonctionnement global de la modélisation des comportements en indiquant comment le fonctionnement des graphes de nœuds permet cette modélisation. Nous expliquons en détail chaque type de graphe et chaque type de nœud pouvant le composer. Nous décrivons comment SAMP-E permet la modélisation de ces graphes de nœuds.

Nous terminons en expliquant comment SAMP-E permet de paramétrer les instances des agents qui seront créées au démarrage des SMA générés.

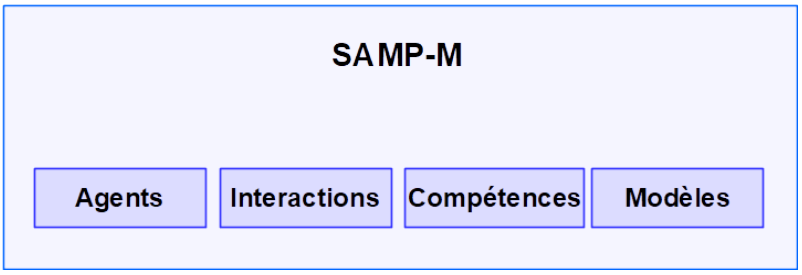

FIGURE 4.1 – SAMP-M est le méta-modèle de SAMP.

4.1/ LES PRINCIPES FONDAMENTAUX DE SAMP

SAMP repose sur des concepts et des approches existants que nous avons assemblés et adaptés afin de répondre au mieux à nos besoins. SAMP doit permettre le développement de jeux vidéo basés sur le paradigme multi-agents. Le fonctionnement des entités présentes dans SAMP doit être abordé sous deux angles différents : celui des agents du paradigme multi-agents et celui de personnages propres au domaine des jeux vidéo.

Dans cette première section de nos contributions, nous abordons les différents concepts sur lesquels nous nous sommes appuyés afin de développer SAMP.

4.1.1/ L'APPROCHE TOUT AGENT POUR UNE PLUS GRANDE FACILITÉ

Nous l'avons vu précédemment, Yoann Kubera [51] a défini une approche tout agent qui consiste à considérer chaque entité comme un agent. Cette approche est très intéressante pour notre objectif de simplicité, car elle permet d'uniformiser toutes les entités du système. Ainsi, les utilisateurs n'ont besoin de paramétrer et modéliser qu'un seul type

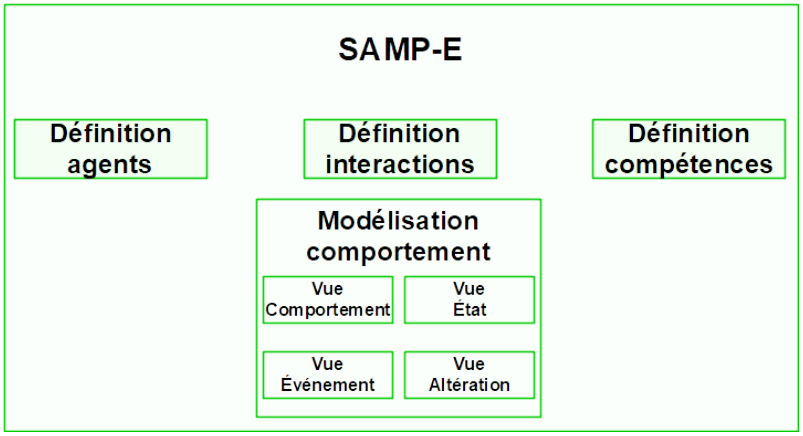

FIGURE 4.2 – SAMP-E est l'outil graphique permettant d'utiliser SAMP.

d'entités. Dans cette approche, chaque agent possède un état (statut [1]) d'activité permettant de définir si un agent est actif, passif ou s'il est capable de changer de statut sans exécuter d'action ou sans être la cible d'une action.

Ainsi, l'approche proposée par SAMP pour générer les SMA repose sur le tout agent. Cependant, nous n'avons pas utilisé le statut de l'approche tout agent de Yoann Kubera de la même manière que lui. Ce statut permettant de savoir si un agent est actif ou passif joue un rôle très important pour l'efficacité du fonctionnement des interactions dans SAMP comme nous le décrivons dans la suite de ce mémoire (section 5.2).

Une des conséquences de l'utilisation de l'approche tout agent est que l'environnement est un agent. Dans SAMP, l'environnement est un agent avec quelques particularités. C'est le seul agent de SAMP qui ne respecte pas le principe d'agents *atomiques* de Jean-Christophe Routier, Philippe Mathieu et Yann Secq [68] que nous abordons dans la section sur les compétences des agents de SAMP. Chaque SMA modélisé et exécuté avec SAMP possède un environnement. Cet environnement permet de faire le lien entre les agents et le moteur physique utilisé (si un moteur physique est utilisé). Il permet par exemple de notifier les agents s'ils sont entrés en contact avec d'autres agents. Il permet aussi aux agents de récupérer une liste des agents à proximité d'eux, de lister les agents en fonction de leur type ou d'une propriété particulière. Cet environnement connaît tous les agents du SMA, il permet la création et la destruction d'agents au cours de l'exécution du SMA.

Nous le verrons dans la suite, lorsqu'un agent demande des informations sur d'autres agents à l'environnement, celui-ci contact les agents cibles afin de savoir s'ils sont d'accord pour transmettre ces informations à l'agent requérant (en se basant sur le principe des facilitateurs).

Le fait d'utiliser l'approche tout agent ne résout pas la duplicité des entités des SMA de SAMP. Dans la suite de ce mémoire, lorsque nous parlons de personnage, il s'agit bien d'agent, mais que nous abordons du point de vue des jeux vidéo. Et pour accompagner au mieux les utilisateurs, nous avons utilisé d'autres concepts du paradigme multi-agents afin de coller au plus près des jeux vidéo.

1. Dans la suite du document, nous utilisons le terme statut afin de ne pas confondre avec les états qui composent le méta-modèle que nous décrivons.

4.1.2/ COLLER AU PLUS PRÈS DES JEUX VIDÉO

En plus de l'approche tout agent permettant de faciliter la compréhension de SAMP par les utilisateurs débutants, nous avons cherché quels concepts très présents dans les jeux vidéo devaient être intégrés à nos travaux.

Dans un jeu vidéo, ce qui différencie en premier les personnages c'est leurs propriétés. On peut comparer deux personnages sur leur force ou leur vitesse en comparant leurs propriétés correspondantes. Afin d'intégrer cette forte dépendance des jeux vidéo aux propriétés des personnages, il nous est apparu que les propriétés des agents de SAMP devaient avoir une importance particulière. Elles doivent pouvoir être utilisées comme des propriétés d'agents (présent au sein d'un SMA), mais aussi être exposées et utilisables au sein du jeu comme des caractéristiques permettant de différencier les personnages.

Les compétences font partie intégrante des jeux vidéo. Elles permettent de créer des différences entre deux personnages ou entités d'un jeu. Les compétences dans les jeux vidéo ne sont pas la spécificité de quelques types de jeu, mais d'une grande majorité. Un grand nombre de types de jeux vidéo mettent en jeu des compétences. On peut citer par exemple :

— Les jeux de rôle où les joueurs peuvent faire évoluer leurs personnages en leur faisant acquérir de nouvelles compétences ou en améliorant celles qu'ils possèdent déjà ;

— Les jeux de sport où les caractéristiques de chaque personnage peuvent lui permettre de réaliser des actions que d'autres personnages ne peuvent pas faire ;

— Les jeux de courses où il est possible de donner des compétences à son véhicule (une bonbonne de nitro pour une accélération puissante par exemple) ;

— Les jeux de stratégie où il est possible d'améliorer chaque unité en lui donnant de nouvelles compétences.

Il nous est apparu que les compétences devaient être un des fondements des agents de SAMP. Qu'elles devaient faire partie, au même titre que les propriétés, des données que les agents exposent afin de pouvoir être utilisées comme des compétences propres aux SMA mais aussi comme des compétences des personnages des jeux.

4.2/ DES COMPÉTENCES...

Nous l'avons vu, les compétences font partie intégrante des jeux vidéo et leur intégration dans SAMP doit aider au fonctionnement des agents au sein des SMA, mais aussi au sein des jeux vidéo.

4.2.1/ L'ACQUISITION DES COMPÉTENCES

En nous inspirant des travaux de Jean-Christophe Routier, Philippe Mathieu et Yann Secq [68], nous avons développé un système de compétences dans SAMP. Ces compétences permettent aux agents d'émettre ou de recevoir des interactions. Dans ces travaux, il est expliqué qu'un agent *atomique* ne possède que deux compétences : une pour interagir et l'autre pour apprendre de nouvelles compétences. Nous avons décidé

de réduire le nombre de compétences des agents atomiques de SAMP en ne leur donnant que la compétence pour apprendre de nouvelles compétences. Cette décision se justifie par l'approche tout agent de notre système. Dans SAMP chaque entité du système est un agent et certains de ces agents pourraient ne pas avoir besoin de la capacité d'interagir avec d'autres agents. De plus, nous le verrons dans la suite de cette section que les interactions peuvent être émises ou perçues par les agents lorsqu'ils possèdent les compétences nécessaires, que ce système est automatisé et qu'il n'existe pas qu'un seul type d'interactions dans SAMP.

Le fait de permettre aux agents d'apprendre de nouvelles compétences permet entre autres :

— D'intégrer une fonctionnalité propre à un grand nombre de jeux vidéo directement dans SAMP. Cette fonctionnalité est celle qui permet de faire évoluer les joueurs ou *PNJ* en leur donnant de nouvelles capacités dans le jeu ;

— De ne pas imposer la création d'un agent pour chaque cas particulier que l'on veut gérer dans le système. Il est possible de modifier le comportement et la capacité d'agents à réaliser certaines actions en ayant des instances différentes d'un même agent.

Par exemple, un agent *rocher* n'aurait pas la capacité d'interagir avec les autres agents. On estime qu'il n'est pas possible dans le système de déplacer, ramasser ou endommager le rocher. Si un agent entre en contact avec le *rocher*, c'est le moteur physique qui transmet les informations à l'agent qui est entré en contact avec lui. Mais si un nouvel agent plus puissant que tous les autres agents du système apparait, il est possible que le rocher puisse être endommagé. Dans ce cas, il est possible de lui offrir une compétence pour interagir avec ce nouvel agent.

Dans SAMP, la manière d'acquérir de nouvelles compétences n'est pas limitée. Il est possible qu'un agent acquière une nouvelle compétence en obtenant un nouvel objet dans son inventaire (une pioche permettant de récolter de la pierre par exemple), en dépensant une monnaie particulière (de l'expérience par exemple) ou en l'acquérant directement.

4.2.2/ DES COMPÉTENCES AVEC DES PRÉ-REQUIS

Une compétence est identifiée par son nom et une liste de t-uplet composés de l'identifiant d'une propriété et d'une valeur minimum. Les agents doivent posséder chaque propriété listée avec une valeur minimum. Celle-ci correspond à la valeur requise pour maîtriser la compétence associée (cf 4.4). Il n'y a aucune restriction au nombre de propriétés requises. La seule restriction est que les propriétés requises doivent avoir une valeur de type primitif (entier, flottant, double, booléen). Dans le cas où une compétence requiert une valeur de type booléen, cela signifie que pour maîtriser cette compétence, l'agent doit posséder la propriété (si le booléen est à *vrai*) ou ne pas posséder cette propriété (si le booléen est à *faux*). Cela pourrait être nécessaire dans le cas où un agent avec la propriété *intangible* voudrait maîtriser une compétence pour attraper des objets. Son intangibilité l'empêcherait de maîtriser cette compétence.

$$Comptence = identifiant, \{\{propriété, niveau\_de\_maitrise\}\}$$

Dans l'exemple, la compétence *se déplacer* requiert, pour être acquise par un agent, qu'il possède la propriété *vitesse* (sans valeur minimum).

$$Se\_déplacer = "seDeplacer", \{\{vitesse, 0\}\}$$

Ces restrictions, faites sur les compétences, permettent avant tout d'éviter de créer des incohérences en empêchant les agents d'acquérir des compétences pour lesquelles ils n'ont pas les pré-requis. Les jeux vidéo sont principalement basés sur leurs règles, celles qui régissent l'univers dans lequel le jeu se déroule, mais aussi les règles du jeu à proprement parler. Le système de compétences présent dans SAMP permet de définir une partie des règles du jeu.

Il est à noter que ce système n'amène aucun surcoût pour les agents ne possédant aucune compétence. Nous voyons dans la section suivante (4.3) les impacts de ce système, couplé au système d'interactions, sur les ressources.

4.2.3/ DÉFINITION DES COMPÉTENCES DANS SAMP-E

L'utilisation de SAMP-E permet de définir ces règles du jeu directement depuis une interface graphique permettant de ne pas avoir recours à du code informatique. La figure 4.3 montre la fenêtre de définition des compétences dans SAMP-E. Dans cette figure, nous voyons que la fenêtre se divise en deux parties. La première, celle de gauche, liste toutes les compétences que les agents peuvent maîtriser. Il est possible de créer et supprimer les compétences. Lorsqu'une compétence est sélectionnée dans cette liste, l'interface se met à jour afin d'afficher les données relatives à cette compétence dans la deuxième partie. Cette deuxième partie liste les propriétés qu'un agent doit posséder pour pouvoir maîtriser la compétence sélectionnée. Dans notre exemple, la propriété *See* requiert, de la part des agents voulant la maîtriser, qu'ils possèdent une propriété *Vision*. On constate que dans notre exemple, la compétence *See* requiert que les agents aient la propriété *Vision* avec une valeur minimale de 0. Cela signifie que n'importe quel agent possédant cette propriété sera capable de maîtriser la compétence *See*. Nous aurions pu, dans ce cas particulier, requérir pour la compétence *See* une propriété *Vision* de type booléenne. Dans ce cas, si un agent ne possède pas la propriété *Vision* ou si la valeur de cette propriété est égale à *faux*, alors l'agent ne serait pas dans la capacité de maîtriser la compétence *See*.

Dans l'exemple proies-prédateurs, les loups et les moutons possèdent les compétences pour se déplacer, être vus, voir, attaquer et se reproduire. L'herbe, quant à elle, possède les compétences pour être vue et coloniser du terrain vierge. Les moutons et l'herbe possèdent une compétence pour être attaqués. La table 4.1 résume l'ensemble des compétences nécessaires à l'exemple.

Nous allons voir dans la suite de ce chapitre que l'acquisition de compétences offre automatiquement la possibilité d'émettre ou de recevoir des interactions et qu'il est possible de paramétrer des agents pour qu'ils possèdent des compétences dès leur initialisation.

4.3/ ...ET DES INTERACTIONS

Dans la suite de ce mémoire, nous considérons qu'une interaction n'est pas seulement un acte de langage (communication), mais plus généralement un échange d'informations entre deux agents.

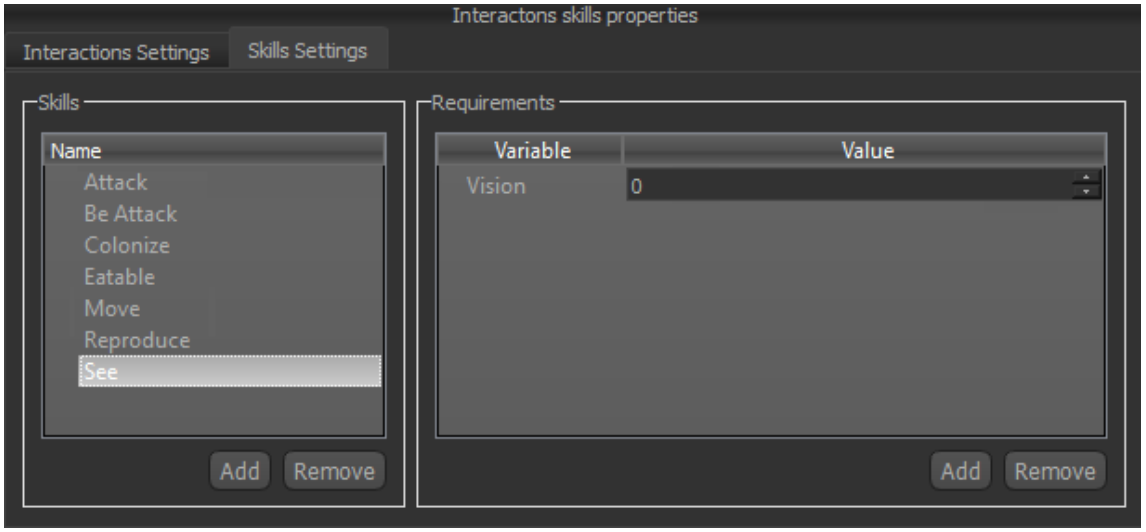

FIGURE 4.3 – Fenêtre de définition des compétences dans SAMP

Les interactions sont les actions qu'un agent est capable d'émettre vers un autre agent. Dans les jeux vidéo, les interactions sont omniprésentes : lorsqu'un personnage en touche un autre, lorsqu'un personnage voit ou entend un autre personnage, il y a interaction. Si on pousse le concept d'interaction au maximum, lorsque le joueur utilise un périphérique, il émet une interaction vers le personnage qu'il contrôle. Mais les interactions permettent aussi de façonner le jeu, de le rendre plus réaliste. Quand un joueur peut ramasser un morceau de bois, détruire une caisse ou pousser une table, il s'immerge plus facilement dans l'univers dans lequel évolue son personnage.

Afin de permettre aux utilisateurs de SAMP de plus facilement gérer ses interactions, nous avons développé un système d'interactions automatisées couplé avec le système de compétences précédemment décrit. Dans cette section, nous décrivons ce système d'interactions.

4.3.1/ UNE AUTOMATISATION GRÂCE AUX COMPÉTENCES

Contrairement à IODA [52] qui est un système orienté interactions, nous avons un système orienté états/événements. Bien que nous nous soyons inspirés de ces travaux, nous n'avons pas pris le parti de définir chaque action des agents comme étant des interactions. Dans IODA, lorsqu'un agent se déplace, il s'agit d'une interaction entre lui et l'environnement. Ce fonctionnement n'est cependant pas adapté à l'utilisation combinée avec un moteur physique. Dans SAMP, lorsqu'un agent se déplace, c'est le moteur physique qui modifie la position de l'entité physique représentant l'agent.

Nous avons fixé le fait que dans SAMP les interactions sont tous les échanges d'informations entre différents agents ou entre un agent et l'environnement. Que ce soit un échange d'informations volontaire comme lors d'une communication verbale ou lors d'un échange non-volontaire comme lorsqu'un agent est vu par un autre agent. Afin d'éviter toute ambiguïté, nous définissons le terme *perception* représentant cette deuxième catégorie d'interaction :

Définition (Perception)**.** *Une perception est tout échange d'informations durant lequel*

TABLE 4.1 – Table des compétences nécessaire au fonctionnement de notre exemple.

Compétence	Propriétés requises
Se déplacer	Vitesse = 0
Être vu	—
Voir	Vision = 0
Attaquer	—
Être attaqué	Résistance = 0
Se reproduire	—
Coloniser	—
Comestible	—

tous les agents concernés ne sont pas conscients de cet échange.

Par exemple, lorsqu'un agent A est vu par un agent B, l'agent A n'est pas conscient qu'il transmet des informations à B.

Dans SAMP, afin de faciliter la modélisation des SMA, nous avons cherché à automatiser le fonctionnement des interactions. Nous l'avons déjà vu, les agents de SAMP peuvent posséder des compétences qui leur permettent d'exécuter de nouvelles actions. Ces actions peuvent être l'émission ou la réception d'interactions et ainsi, l'acquisition de nouvelles compétences peut permettre à un agent d'émettre ou recevoir des interactions.

Cette automatisation possède un coût dû au fait qu'une recherche est effectuée dès qu'une modification est exécutée sur une compétence d'un agent. Le coût maximum de cette recherche est :

Posons :

- Γ l'ensemble des interactions du système ;
- Γ_n une interaction du système tel que $\Gamma_n \in \Gamma$;
- $Competence(\Gamma_n)$ la fonction donnant le nombre de compétences requises par Γ_n ;
- Alors, le coût maximum de la recherche des interactions pour un agent est :

$$\sum_{int=0}^{int=|Interactions|} Comptence(\Gamma_n) \tag{4.1}$$

Pour minimiser ce coût et éviter que le parcours soit exécuté dès qu'une compétence d'un agent est modifiée, nous avons réduit le nombre de parcours nécessaire à 1 par frame. Chaque agent garde une information permettant de savoir, si au cours d'une frame, il a acquis une compétence et une information permettant de savoir s'il a perdu une compétence. Ainsi, à la fin de la phase de *PostUpdate* (cf 2.1.3), si l'agent a acquis ou perdu une ou plusieurs compétences, il n'y a qu'un unique parcours qui exécute trois types de recherches différentes :

- Si l'agent a acquis une ou plusieurs compétences, il n'exécute la recherche que sur les interactions qu'il ne peut pas déjà émettre ou recevoir afin de trouver de nouvelles interactions qu'il peut émettre ou recevoir ;
- Si l'agent a perdu une ou plusieurs compétences, il n'exécute la recherche que sur les interactions qu'il peut déjà émettre ou recevoir afin de vérifier quelles interactions il ne peut plus émettre ou recevoir ;

— Si l'agent a acquis et perdu des compétences, les deux points précédents sont exécutés, mais toujours à l'aide d'un seul parcours.

Le fait de réaliser cette action à la fin de la phase de *PostUpdate* permet de s'assurer que tous les agents acquièrent la capacité de recevoir et émettre des interactions au même moment que les autres agents.

Dans notre système, si une interaction ne requiert pas de compétence pour être reçue (resp. émise) c'est qu'elle peut être reçue (resp. émise) par tous les agents du système. Dans SAMP, lorsqu'un agent possède les compétences nécessaires pour émettre (resp. recevoir) une interaction, il est automatiquement capable d'émettre (resp. recevoir) cette interaction.

Cette automatisation permet aux utilisateurs de SAMP d'avoir uniquement à gérer les compétences que les agents acquièrent ou perdent sans avoir, en plus, à se préoccuper des interactions. Il est à noter qu'il n'y a pas de limitations dans ce système d'interactions et compétences. Une interaction peut nécessiter un nombre indéfini de compétences pour être émise ou reçue et une compétence peut-être utile à un nombre indéfini d'interactions.

4.3.2/ UNE STRUCTURE DE DONNÉES MINIMALE

Nous avons voulu la structure des interactions la plus petite possible afin d'éviter de consommer trop de ressources. La structure que nous proposons est la plus minime possible pour fonctionner dans SAMP. Les interactions de SAMP possèdent toutes les mêmes propriétés :

— un identifiant ;

— deux listes de couples formés d'une compétence et du niveau de maîtrise de cette compétence requis pour exécuter les interactions. Ces listes permettent d'identifier les compétences nécessaires pour recevoir et émettre l'interaction ;

— une liste de variables pouvant être nécessaire au fonctionnement de l'interaction. Cette liste de variables est initialisée par les agents émetteurs au moment de l'émission de l'interaction.

Dans notre exemple, l'interaction *Mange* serait ainsi paramétrée :

Mange = ("Attaque", {{Attaquer, 0}}, {{Être attaqué, 0}}, {Force})

Les valeurs de maîtrise sont à 0, car l'interaction *Attaque* requiert les compétences *Attaquer* et *Être attaqué* sans niveau minimum de maîtrise. L'interaction requiert en plus une variable que nous avons appelée *Force* correspondant à la force de l'attaque exécutée par l'agent émetteur. Le fait que ce soit l'agent émetteur de l'interaction qui calcule et transmette la valeur de la variable de force permet de gérer différentes situations particulières. La liste suivante n'est pas exhaustive et s'appuie sur des exemples particuliers et non sur l'exemple fil rouge :

— Si un agent attaquant possède un bonus particulier permettant d'augmenter temporairement sa force ;

— Si un agent attaquant possède deux armes et attaque à l'aide de ces deux armes. Le fait de calculer la force une seule fois permet de n'émettre qu'une seule fois l'interaction.

Toutes les interactions n'ont pas nécessairement besoin de variables. Une liberté totale est laissée aux utilisateurs afin que les interactions transmettent les informations minimales requises pour leur fonctionnement. Les variables transmises avec les interactions ne sont pas des propriétés que les agents doivent posséder, ce sont des valeurs que les agents transmettent lors de l'émission de l'interaction. Ces valeurs peuvent refléter des propriétés d'un agent, mais aussi être calculées ou générées à partir d'autres informations. Dans notre exemple, les moutons n'ont pas de force. Ainsi, lorsqu'un mouton va émettre une interaction *Mange*, il transmettra une force à une valeur de 0.

La table 4.2 représente les interactions que les agents peuvent émettre ou recevoir dans le système défini par l'exemple proies-prédateurs. Dans cette table, on constate que les interactions *Se déplace* et *Colonise* peuvent être reçues par chaque agent. Dans les faits, il est possible que l'herbe colonise un agent mouton par exemple. C'est la modélisation du comportement de l'herbe qui permettra de fixer son comportement pour l'empêcher (ou l'autoriser selon nos envies) de coloniser un mouton. L'environnement étant un agent comme un autre, il est capable de recevoir les interactions *Colonise* et *Se déplace*.

TABLE 4.2 – Table des interactions des agents du système proies-prédateurs

Interaction	**Émettre**	**Recevoir**	**Valeurs**
Colonise	Coloniser	–	–
Mange	Manger	Comestible	–
Voit	Visible	Voir	–
Se déplace	Se déplacer	–	–
Attaque	Attaquer	Être attaqué	Force

Lors de l'exécution du système, les agents peuvent émettre ou recevoir des interactions. Chaque interaction émise sera formée de la même manière afin de pouvoir être lue par tous les agents. Chaque interaction émise sera composée avec :

— un identifiant (sous forme d'un entier) ;

— le pointeur de l'agent émetteur ;

— l'ensemble des valeurs supplémentaires requises.

Ainsi, chaque agent recevant une interaction est capable, grâce à l'identifiant de l'interaction, de savoir s'il peut recevoir ou non cette interaction et est capable d'agir en conséquence. Le fait que chaque interaction émise possède la même forme, l'agent sera capable de gérer n'importe quelle interaction qu'il est en capacité de recevoir. Comme pour la structure de l'interaction elle-même, nous avons mis en place une structure de données pour les envois et réceptions d'interactions la plus petite possible. Dans ce format, la structure est minimale. L'ensemble des valeurs supplémentaires est un simple tableau. Dans SAMP-E les utilisateurs se serviront des identifiants des valeurs pour modéliser les accès à ces valeurs. Dans SAMP-X, comme nous maîtrisons la génération du code, un simple tableau de valeurs est suffisant, car nous savons quelle valeur se trouve à quel indice du tableau. La figure 4.4 explique ce principe. Le tableau en haut de cette figure montre le tableau des identifiants des valeurs. SAMP-E connaît l'indice de chaque identifiant. Lorsqu'on génère le code correspondant, nous ne générons pas les identifiants mais les valeurs directement (le tableau du bas de la figure). Les valeurs dans le tableau se trouvent au même indice que l'identifiant qui leur correspond dans SAMP-E. Ainsi, nous ne surchargeons pas les envois avec des valeurs inutiles comme des identifiants pour ces valeurs supplémentaires.

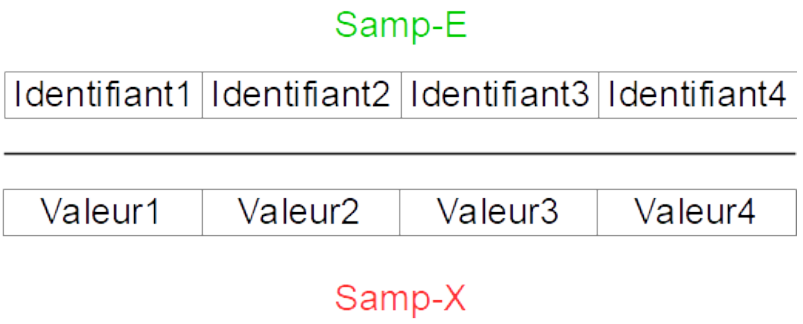

FIGURE 4.4 – Tableaux de valeurs des interactions dans SAMP-E et SAMP-X.

4.3.3/ DÉFINITION DES INTERACTIONS DANS SAMP-E

Tout comme pour les compétences, il est possible de définir les interactions par le biais d'une interface graphique. La figure 4.5 montre l'interface de définition des interactions dans SAMP-E avec les valeurs correspondantes à notre exemple. Nous constatons que cette fenêtre se découpe en 3 parties :

1. La partie à gauche, *Interactions*, énumère toutes les interactions que les agents pourront émettre ou recevoir. Il est possible de créer et supprimer les interactions. Lorsqu'une interaction est sélectionnée, l'interface se met à jour pour afficher les données relatives à cette interaction ;

2. La partie la plus à droite, *Available Skills* liste toutes les compétences créées dans la fenêtre de définition des compétences présentée dans la figure 4.3 ;

3. La partie centrale, *Requirements*, liste, pour l'interaction sélectionnée, les compétences requises pour pouvoir émettre ou recevoir l'interaction. Si aucune compétence n'est indiquée pour émettre (resp. recevoir) une interaction, cela signifie que tous les agents peuvent émettre (resp. recevoir) cette interaction.

Nous l'avons vu précédemment, il est possible d'offrir aux agents la capacité d'émettre ou recevoir des interactions en leur donnant les compétences requises pour le faire. Nous définissons dans la section suivante les agents de SAMP.

4.4/ DÉFINITION DES TYPES D'AGENTS

Dans SAMP, les agents possèdent des propriétés, des compétences et sont capables d'émettre et recevoir des interactions. Cette section décrit comment nous avons développé les agents dans SAMP qui utilisent une approche tout agent. Dans cette section, nous expliquons le fonctionnement des propriétés des agents et nous expliquons comment est géré le système de compétences et d'interactions au sein des agents.

Il est important de préciser que dans cette section, nous parlons des agents en tant que ressources et non en tant qu'instance. Dans SAMP, une instance d'agent est créée à partir des paramètres d'une ressource agent.

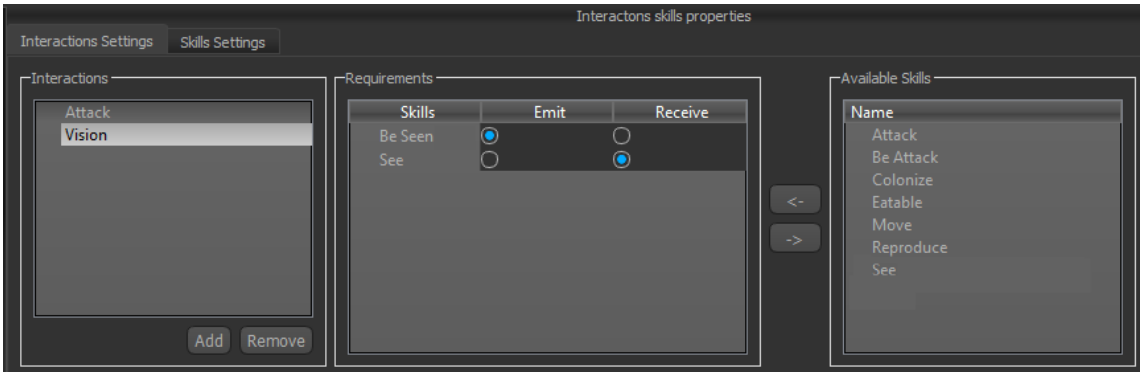

FIGURE 4.5 – Fenêtre de définition des interactions dans SAMP

4.4.1/ LES FACILITATEURS

Tout comme les compétences et les interactions, les propriétés des personnages dans les jeux vidéo sont omniprésentes. Les propriétés définissent si un agent est capable ou non d'exécuter une action et avec quelle efficacité. Que ce soit dans des jeux de sport où les propriétés déterminent l'habilité d'un joueur ou dans des jeux de rôle où les propriétés déterminent la force des coups et la résistance à ces coups de deux personnages se battant.

SAMP met à disposition des agents possédant des propriétés qu'il est possible de paramétrer entièrement. Chaque agent possède une liste de propriétés qui peuvent être de n'importe quel type. Il est possible de donner une valeur par défaut pour les propriétés d'une ressource agent afin que, lors de l'instanciation d'un agent, cette propriété possède cette valeur. Les propriétés des agents peuvent être utilisées et modifiées lors de l'exécution de leur comportement.

Nous l'avons vu dans la section 2.2.4, FIPA a défini des agents particuliers appelés *facilitateurs* permettant de fournir aux agents la liste des agents avec lesquels ils peuvent communiquer (Agent Management System - AMS) et la liste des services proposés par ces agents (Directory Facilitator - DF). Les *facilitateurs* de FIPA sont principalement pensés pour le monde de l'industrie. Dans ce domaine, les agents ont un but commun, faire avancer le système. Les agents ne mentent pas et collaborent sans se préoccuper de leurs intérêts personnels. Dans les jeux vidéo, c'est différent. Les agents peuvent avoir des objectifs personnels, ils peuvent décider de mentir à d'autres agents, de ne pas collaborer voire même de saboter le travail d'un autre. Pour ces raisons, nous nous sommes inspirés des *facilitateurs* de FIPA, mais avons dû appliquer quelques changements.

Tout d'abord, nous l'avons vu dans la section 2.2.4, FIPA est basé sur 5 niveaux. Dans nos travaux, nous n'utilisons pas le niveau *langage de contenu* qui n'a pas d'utilité dans un système fermé comme l'est un jeu vidéo. Nous avons aussi modifié le niveau *ontologie* en définissant une seule ontologie avec la possibilité de spécialiser cette ontologie avec de nouveaux vocabulaires que certains agents en particulier connaîtraient.

Pour ce qui est de l'AMS qui, rappelons le, permet d'indiquer aux agents la liste des agents avec qui ils peuvent communiquer, mais aussi qui sont responsables de la création et la destruction des agents, nous avons décidé de l'intégrer à l'agent environnement des SMA SAMP.

Pour ce qui est des DF, qui ont pour rôle d'indiquer aux agents les services proposés

par les autres agents, nous avons décidé de donner ce rôle à chaque agent du système. Cela permet d'offrir aux agents un comportement plus proche de la réalité en leur offrant la capacité de décision quant aux demandes qui leurs sont faites. Ainsi, lorsqu'un agent demandera à un autre agent la valeur d'une propriété ou de venir l'aider sur une situation requérant une coopération, l'agent sollicité, par le biais de son DF, prendra une décision : répondre positivement ou négativement à la demande. Il sera même possible aux agents de fournir une réponse erronée que ce soit volontaire ou non.

Pour les demandes de coopération, l'agent recevra une interaction contenant les informations de cette demande. Le DF de l'agent décidera si, oui ou non, il accepte de venir en aide à l'agent requérant et modifiera son comportement en fonction. Cette interaction possédera l'identifiant *cooperate*. L'objectif de la coopération sera défini dans les valeurs supplémentaires de l'interaction. Chaque utilisateur pourra ainsi définir les objectifs comme il le veut.

Une coopération peut prendre plusieurs formes. Cela peut-être une coopération où un agent requiert une action d'un autre agent ou simplement une coopération où un agent requiert une information détenue par un autre agent. Comme dans la réalité, un agent requis pour une coopération peut refuser cette coopération et ce refus peut prendre plusieurs formes :

— L'agent décide de ne pas coopérer volontairement et préviens l'agent requérant qu'il refuse ;

— L'agent décide de ne pas coopérer volontairement, mais ment sur ses intentions (en donnant une information qu'il possède, mais avec une valeur erronée) ;

— L'agent exprime son envie de coopérer, mais sabote la coopération. Similaire au fait que l'agent mente. Mais dans le cas du mensonge, l'agent requérant ne sait pas qu'il y a eu un mensonge. Dans le cas du sabotage, l'agent requérant sait qu'il y a eu sabotage, car la coopération a échoué par la volonté de l'agent requis.

La décision qu'un agent prend, lorsqu'une demande de coopération lui est transmise, est prise par son *DF*.

Lorsqu'un agent a besoin de connaître la valeur d'une propriété d'un autre agent, il envoie une requête à cet agent. Le DF prend alors la décision de répondre positivement ou non à cette requête. Le fait que les agents aient chacun le contrôle sur ce qu'ils exposent ou non permet de coller au plus près de la réalité. Il était important que ce soit uniformisé pour ne pas alourdir la compréhension aux utilisateurs.

L'avantage de cette méthode est qu'en laissant une totale liberté de définition aux utilisateurs sur ce qu'un agent accepte de partager ou non, permet de modéliser tous les comportements imaginables sur les réactions qu'un agent aura face à ce genre de requête. Le risque est qu'une mauvaise modélisation amènera un comportement irréaliste avec des agents refusant de partager des propriétés pourtant accessibles aux yeux de tous comme la couleur de leurs cheveux, leur taille ou s'ils portent un sac par exemple.

Chaque agent possédant son propre DF, le risque de créer une file d'attente de requêtes sur un seul agent DF est limité. De plus, il est plus simple de créer, à partir d'un même agent, des instances qui prendront des décisions différentes.

Nous allons maintenant expliquer comment donner ou retirer des compétences aux agents et l'impact que cela a sur eux.

4.4.2/ ACQUISITION OU PERTE DE COMPÉTENCES ET INTERACTIONS

En se basant sur les agents atomiques définis par Jean-Christophe Routier, Philippe Mathieu et Yann Secq dans [68], nous avons donné aux agents la possibilité d'acquérir des compétences (définies dans 4.2). Il est possible de paramétrer une ressource agent avec des compétences de bases afin que les instances de cette ressource possèdent, dès leur création, ces compétences.

Il est aussi possible que les agents acquièrent de nouvelles compétences durant leur cycle de vie. Pour acquérir une compétence, chaque agent possède une méthode lui permettant d'ajouter une compétence à la liste de celles qu'il possède déjà. Cette méthode prend en argument l'identifiant de la compétence et retourne un booléen indiquant si, oui ou non, la compétence a été acquise par l'agent.

Il est possible qu'un agent ne puisse pas acquérir une compétence si, par exemple, il ne possède pas les propriétés nécessaires. Un agent est capable d'acquérir des compétences de plusieurs manières différentes. La liste suivante n'est pas exhaustive, chaque utilisateur pourra définir les différentes façons de faire acquérir de nouvelles compétences à un agent :

— par acquisition directe. Il s'agit d'une acquisition qui se fait directement, par exemple lorsque l'agent a atteint un niveau suffisant ou qu'une monnaie a été dépensée pour lui faire acquérir cette compétence ;

— par transmission. Dans ce cas, l'agent a appris, par le biais d'un autre agent, à maîtriser la compétence ;

— par acquisition indirecte. Il s'agit d'une méthode dans laquelle l'agent acquiert la maîtrise d'une compétence par le biais de l'acquisition d'un autre agent. Si un agent guerrier ramasse une épée (qui est un agent dans l'approche tout agent), il possédera alors la compétence pour attaquer à l'épée. Cela se rapproche de la méthode par transmission dans le sens où c'est l'agent épée qui *transmet* à l'agent guerrier la compétence pour attaquer à l'épée.

Cependant, si les agents sont capables d'acquérir de nouvelles compétences, ils sont aussi capables de perdre des compétences. De la même manière que pour l'acquisition d'une compétence, un agent possède une méthode qui permet de lui faire perdre une compétence. Cette méthode prend en argument l'identifiant de la compétence à perdre et retourne un booléen indiquant si, oui ou non, la compétence a été perdue par l'agent.

En reprenant l'exemple de l'agent acquérant une compétence par le biais d'un agent épée, s'il perd cette épée, il perdra la compétence pour attaquer à l'épée.

Lorsqu'un agent gagne ou perd une compétence, il vérifie quelles interactions il peut ou ne peut plus émettre et recevoir. Tout ceci est géré automatiquement. Seuls l'ajout ou le retrait des compétences doivent être modélisés par l'utilisateur.

Cette possibilité offerte aux utilisateurs d'ajouter ou retirer des compétences à un agent et de définir les différentes manières que les agents ont de gagner ou perdre ces compétences permet de modéliser des comportements très différents pour l'acquisition et la perte de compétences.

L'approche tout agent ([51]) associée aux agents atomiques ([68]) permet de n'avoir qu'un seul type d'entité dans nos systèmes et que ces entités soient capable de se différencier, avant même qu'on leur applique un comportement, sur leurs capacités. On s'approche

d'un système anthropomorphique où des individus peuvent être comparés sur ce qu'ils sont capables de faire avant d'avoir à le faire.

Comme nous l'avons déjà expliqué dans la section 4.3, l'ajout ou la perte de compétence ne créés pas de surcoût. C'est la gestion des interactions qui suit ces modifications sur les compétences qui créé un surcoût que nous maîtrisons.

Ce fonctionnement permettant d'acquérir la possibilité d'émettre et recevoir des interactions en fonction des compétences maîtrisées par les agents permet de se rapprocher du concept d'Affordance de James Gibson [40]. Ce concept expose que des informations des actions possibles sont présentes dans l'environnement. Ce concept a été appliqué dans le domaine des SMA [5]. Le principe est de permettre à des agents d'interagir avec d'autres agents en fonction de leur type.

Dans les SMA, l'utilisation des affordances permet, par exemple, de différencier les interactions qu'un agent *homme* et un agent *fourmi* peuvent échanger avec un agent *chaise*.

Dans SAMP le fait que les agents acquièrent la capacité d'émettre ou recevoir des interactions permet de *simuler* une partie du concept d'affordances. En gardant cet exemple d'agents *homme*, *fourmi* et *chaise*, les affordances devraient permettre de différencier le fait qu'un homme peut s'asseoir sur la chaise et la fourmi grimper sur la chaise. Dans SAMP, nous pouvons créer une interaction *grimper_sur_chaise* et une interaction *asseoir_sur_chaise*. Ensuite, il suffirait de donner les compétences à la chaise pour recevoir ces deux interactions et les compétences pour émettre la compétence *asseoir_sur_chaise* à l'agent *homme* et les compétences pour émettre l'interaction *grimper_sur_chaise* à l'agent fourmi. De cette manière, chaque agent pourra émettre une interaction différente vers l'agent *chaise*.

4.4.3/ DÉFINITION DU COMPORTEMENT DES AGENTS

Les agents peuvent maîtriser des compétences et émettre ou recevoir des interactions. Mais ils peuvent surtout exécuter des comportements. Ces comportements peuvent être appliqués aux agents afin que ces derniers exécutent les instructions du comportement. Toujours dans le but de faciliter la modélisation des SMA développés avec SAMP, nous avons cherché à rendre le plus intuitif possible la définition des comportements.

Tout comme les agents, les comportements de SAMP possèdent des propriétés permettant de différencier deux exécutions d'un même comportement. Nous le voyons dans la suite de ce chapitre, ces propriétés sont initialisées lorsque les comportements sont initialisés.

En plus des propriétés, les comportements possèdent des restrictions. Comme pour les compétences des agents, ces restrictions permettent d'empêcher certains agents d'exécuter des comportements particuliers. Les restrictions sur les comportements vérifient les compétences et les propriétés que les agents possèdent. Une vérification est faite sur les propriétés afin qu'un comportement se servant, par exemple, de la propriété *vitesse* d'un agent ne soit pas exécuté par un agent n'ayant pas cette propriété. Pour cette même raison, les compétences sont vérifiées afin qu'un agent n'ayant pas la compétence *attaquer* ne puisse pas exécuter un comportement d'attaque.

Chaque comportement possède une liste composée de t-uplet comprenant le nom de la propriété et son type.

Dans notre exemple, les agents voulant exécuter le comportement des loups doivent posséder les propriétés *Niveau de Faim*, *Seuil de Faim*, *Perte de Faim à l'arrêt* et *Perte de Faim en déplacement* chacune de type entier.

Si un agent possède les propriétés pour exécuter un comportement, il est possible de lui assigner ce comportement directement depuis l'interface graphique de SAMP-E.

Nous avons abordé et défini le fonctionnement et la définition des compétences, des interactions et des agents. Nous allons maintenant expliquer comment modéliser le comportement des agents au sein de SAMP.

4.5/ COMPORTEMENTS DES AGENTS

Nous venons de voir comment paramétrer les agents dans SAMP. Nous allons maintenant aborder le comportement de ces agents. Nous avons différencié deux types de comportements qu'un agent pouvait avoir : un comportement de réaction et un comportement d'altération que nous décrivons plus en détail dans la suite de cette section. Ces deux comportements différents peuvent être exécutés en même temps par un agent.

Dans cette section, nous décrivons chacun de ces deux types de comportements en commençant par décrire le fonctionnement des comportements de réaction basés sur un système d'états et d'événements. Nous continuons en expliquant le principe des états à multi-niveaux et nous terminons en décrivant les comportements d'altérations.

4.5.1/ UN SYSTÈME D'ÉTAT ET D'ÉVÉNEMENTS

Les agents de SAMP suivent un comportement basé sur des états et des événements. A chaque instant de sa vie, un agent est dans un état. Chaque état est modélisé par une série d'instructions qui permettent de définir le comportement qu'auront les agents se trouvant dans cet état. Tant qu'un agent est dans un état, il exécute en boucle le comportement modélisé par cet état. Ce comportement à base d'états et d'événements est ce que nous avons appelé le comportement de réaction. L'agent **réagit** à des événements qui le font changer d'état.

Un agent ne peut se trouver que dans un seul état à un instant *t*. Il est capable de changer d'état au cours de son cycle de vie. Un changement d'état s'opère lorsqu'un événement est déclenché. Enfin, il est possible de passer des paramètres en entrée d'un état au moment où un agent entre dans cet état. Cela permet de modifier le fonctionnement d'un état.

Cette approche d'états et d'événements est similaire à un *système à événements discrets* (DEVS [35] par exemple). La différence réside principalement dans le fait que dans un *système à événement discret*, les états ont une durée de vie qui, lorsqu'elle est atteinte, oblige le système à changer d'état. De plus, les états suivants, lors d'une transition, dépendent du temps écoulé dans l'état courant. Dans notre approche, les états n'ont pas de durée de vie et le temps écoulé durant un état n'a pas d'impact sur la désignation de l'état suivant.

Ce principe d'état permet d'économiser des ressources. En effet, à chaque frame, un agent teste si un événement s'est déclenché pour le faire changer d'état. Lorsqu'un agent

est dans un état, il connaît les événements qui pourraient le faire quitter cet état. Ainsi, un agent ne va tester que les événements en lien avec l'état dans lequel il se trouve au moment des tests.

Mais ce système d'état et d'événement permet aussi de gérer une partie des règles du jeu. Si un agent est incapable de réaliser une action (commandée par le joueur) lorsqu'il est dans un état particulier, lorsque le joueur va déclencher l'événement pour réaliser cette action interdite (en appuyant sur un bouton de la manette par exemple), cet événement ne sera pas du tout récupéré par l'agent.

Les événements peuvent prendre plusieurs formes :

- Une interaction venant d'un autre agent ou de l'environnement ;
- La modification de la valeur d'une propriété de l'agent ;
- Lorsqu'un état a fini de s'exécuter. Certains états modélisent un comportement qui a une durée définie. Cela peut être le cas lorsqu'un état joue une animation ;
- Lorsque le joueur transmet une instruction à son personnage.

Plusieurs événements peuvent être liés à un même état et mener, lors de leur activation, à des états différents ou identiques.

Définition (Événements concurrents)**.** *Des événements sont appelés concurrents s'ils sont déclenchés en même temps (durant la même frame) et qu'ils sont reliés à l'état courant d'un agent.*

Le fait qu'un agent ne puisse être que dans un seul état à la fois impose que lorsque des événements sont concurrents, il soit nécessaire de faire un choix. Pour se faire, si plusieurs événements sont liés à un même état, un ordre de priorité sera donné à ces liaisons afin d'ordonner l'ordre d'analyse des événements et ainsi gérer une éventuelle concurrence entre ces événements.

Les tables 4.3, 4.4 et 4.5 donnent les différents états dans lesquels peuvent se trouver les loups, les moutons et l'herbe pour notre exemple. Ces tables donnent aussi les événements permettant de passer d'un état à un autre.

Nous constatons qu'un même état peut servir pour réaliser des tâches similaires avec des comportements différents. Par exemple, l'état *Recherche* est le même que ce soit pour rechercher de la nourriture ou un partenaire pour se reproduire. Seuls les paramètres en entrée de l'état permettront de déterminer la cible qui sera recherchée. Cela permet de factoriser les comportements.

Il est utile aussi de préciser la raison qui amène un agent loup ou un agent mouton à quitter l'état *Recherche* pour entrer à nouveau dans l'état *Recherche* lorsque la propriété de faim atteint un certain seuil. Cela arrive lorsque l'agent *loup* ou l'agent *mouton* est en recherche d'un partenaire pour la reproduction. Il quittera l'état de *Recherche* d'un partenaire pour entrer dans un état de *Recherche* de nourriture.

Nous voyons dans la sous-section suivante qu'il est possible de créer des états à partir d'autres états.

4.5.2/ DIFFÉRENTS NIVEAUX D'ÉTAT

SAMP permet à des utilisateurs sans connaissance des logiques de développement informatique de modéliser des SMA. Mais SAMP permet aussi à des utilisateurs maîtrisant

TABLE 4.3 – Table d'états des agents Loup et événements permettant de changer d'état

		Cible		
		Déplacement	Recherche	Manger
Source	Déplacement	—	Faim < Seuil	—
	Recherche	—	Faim < Seuil	—
	Manger	Reproduction = 0	Reproduction = 1	—
	Se reproduire	Animation terminée	—	—
	Attaquer	—	—	Mouton tué
	Mort	—	—	—

		Cible		
		Se reproduire	Attaquer	Mort
Source	Déplacement	—	—	Faim = 0
	Recherche	Loup trouvé	Mouton Trouvé	Faim = 0
	Manger	—	—	Faim = 0
	Se reproduire	—	—	Faim = 0
	Attaquer	—	—	Faim = 0
	Mort	—	—	—

TABLE 4.4 – Table d'états des agents Mouton et événements permettant de changer d'état

		Cible		
		Déplacement	Recherche	Manger
Source	Déplacement	—	Faim < Seuil	—
	Recherche	—	Faim < Seuil	—
	Manger	Reproduction = 0	Reproduction = 1	—
	Se reproduire	Animation terminée	—	—
	Attaquer	Animation terminée	—	—
	Mort	—	—	—

		Cible		
		Se reproduire	Attaqué	Mort
Source	Déplacement	—	—	Faim = 0
	Recherche	Mouton trouvé	Herbe Trouvée	Faim = 0
	Manger	—	—	Faim = 0
	Se reproduire	—	—	Faim = 0
	Attaquer	—	—	Faim = 0
	Mort	—	—	—

TABLE 4.5 – Table d'états des agents Herbe et événements permettant de changer d'état

		Cible		
		Coloniser	Attente	Mort
Source	Coloniser	—	—	Mangé
	Attente	Colonisation != 0	—	Mangé
	Mort	—	—	—

la logique du développement informatique de modéliser des comportements plus complexes grâce à une grande expressivité. Pour offrir cette simplicité d'utilisation, en conser-

vant une grande expressivité, SAMP offre la possibilité de modéliser des états contenant eux-mêmes des états.

Dans notre exemple, les agents *loup* et *mouton* ont un état appelé *Déplacement* qui leur permet de se déplacer aléatoirement dans l'environnement. Cet état est lui-même composé d'un état *Se Déplacer Vers* qui prend en entrée la destination cible. Dans l'état *Déplacement*, un tirage aléatoire est fait pour obtenir la destination cible qui est passée en paramètre de l'état *Se Déplacer Vers*. Une fois la destination atteinte, l'état *Se Déplacer Vers* émet un événement qui a pour effet qu'une nouvelle destination est tirée aléatoirement avant de retourner dans l'état *Se Déplacer Vers*.

Nous constatons alors deux niveaux de modélisation :

— La modélisation des comportements primitifs par des développeurs expérimentés. Ce niveau de modélisation nécessite de connaître la logique de développement : compréhension des boucles, des conditions, des variables, ...

— La modélisation des comportements abstraits. Ce niveau requiert une faible connaissance de la logique de développement, mais requiert de maîtriser la logique métier. Il s'agit principalement d'assembler des comportements primitifs entre eux.

Lorsqu'un agent se trouve dans un état lui-même composé d'états, il exécute le comportement de l'état de plus bas niveau (le plus primaire), mais les événements qui permettent de changer d'état sont testés en partant de l'état le plus abstrait. Lorsqu'un agent quitte un état, cet état est *mis en pause* et l'agent pourra revenir dans cet état et reprendre son exécution là où il l'avait arrêtée.

Dans notre exemple, si un agent mouton est dans l'état *Déplacement*, son état courant est en fait l'état *Se Déplacer Vers*. Pour quitter l'état *Se Déplacer Vers*, nous l'avons vu, il faut que l'agent atteigne sa destination. Mais, si l'agent est attaqué par un loup, il quittera l'état *Déplacement* (et par conséquent, il quittera aussi l'état *Se Déplacer Vers*) pour entrer dans l'état *Attaqué*.

Les états comme les événements sont modélisés à l'aide de graphes de nœuds. Dans la suite de cette section, nous décrivons les différents types de modèles (appelés vues) qu'il est possible de créer dans SAMP. Nous décrivons aussi les différents nœuds qui peuvent composer ces modèles.

4.5.3/ LES ALTÉRATIONS

Nous l'avons vu précédemment, le comportement des agents est divisé en deux catégories. Nous avons déjà défini le comportement de réaction des agents. Nous allons aborder le comportement d'altération.

Nous l'avons vu, lorsqu'un agent est dans un état, il ne gère que les événements en lien avec cet état. De plus, si plusieurs événements sont concurrents, l'agent ne gère qu'un seul de ces événements. Afin d'éviter que certains événements impactant l'agent ne soient perdus, nous avons mis en place les altérations. Tous les événements ne sont pas sujets à être gérés dans les altérations. Seuls ceux entraînant un changement des propriétés des agents doivent être gérés dans les altérations.

Définition (Altération)**.** *Une altération est la conséquence d'un événement qui impacte un agent sans que celui-ci y réagisse directement.*

Il est important de bien différencier les altérations des réactions. La conséquence d'une réaction est que l'agent change d'état. La conséquence d'une altération est que l'agent va subir des modifications de propriétés. Il est possible qu'un événement soit à la source d'une réaction et d'une altération. Lorsqu'un agent *loup* attaque un *mouton*, le mouton va réagir en entrant dans un état où il va jouer une animation signifiant qu'il est attaqué. Dans cet état, il ne perdra pas de point de vie. En plus de cette réaction, le mouton va subir une altération que lui fera perdre des points de vie.

Dans notre exemple, le système de faim est une altération. A intervalle régulier, les agents loup et mouton voient leur propriété de *Faim* diminuée. Il s'agit d'une altération dont l'événement est la fin d'un *timer*. Il s'agit d'une altération, car l'agent dont la propriété de *Faim* diminue ne va pas réagir directement à cette diminution. Il réagira au fait que la propriété de *Faim* est passée sous un certain seuil.

Dans son comportement d'altération, du fait qu'un agent ne modifie pas son comportement à cause d'un événement, un agent peut être altéré par plusieurs événements, même concurrents. Le comportement altération ne permet pas aux agents de changer d'état ni d'effectuer des actions autres que des modifications de leurs propriétés.

4.6/ MODÉLISATIONS DES COMPORTEMENTS PAR QUATRE VUES

Dans la section précédente, nous avons décrit les deux types de comportements pouvant être exécutés par les agents de SAMP. Dans cette section, nous allons expliquer comment modéliser ces comportements.

Pour rappel, dans un but de permettre l'utilisation de SAMP à un grand nombre d'utilisateurs, et ce, quelles que soient leurs compétences en développement informatique, la modélisation des comportements se fait par l'utilisation de graphes de nœuds (aussi appelés *vues*). Chaque vue peut être modélisée dans une interface graphique proposée par SAMP-E. Il existe 4 types de vues différentes :

— Les vues de *comportement* qui sont les points d'entrée des comportements de réaction ;

— Les vues d'*état* permettant de modéliser le comportement des états dans lesquels les agents peuvent se trouver ;

— Les vues d'*événement* permettant de modéliser un événement ;

— Les vues d'*altération* permettant de modéliser le comportement des altérations.

Dans cette section, nous commençons par décrire les généralités concernant les vues et les nœuds qui les composent. Nous continuons cette section en définissant le fonctionnement de chaque type de nœud et les liens qui permettent de les unir. Nous terminons en décrivant les 4 vues : d'*altération*, de *comportement*, d'*état* et d'*événements*. Pour chaque type de graphe de nœuds que nous décrivons, nous donnons les spécificités le concernant : nœuds (in)utilisables, sémantique et usage des nœuds.

4.6.1/ LE PARAMÉTRAGE DES VUES

SAMP permet de modéliser les comportements des agents en créant des vues. Chacune de ces vues a un rôle particulier que nous décrivons dans la suite de cette section. Dans

cette sous-section, nous nous attardons sur la manière de paramétrer ces vues. Les vues permettent de faire une projection du méta-modèle. Les paramètres permettent de configurer ces projections.

Ce paramétrage possède plusieurs buts :

1. Ajouter des variables qui seront accessibles dans chaque instance de la vue. Ces variables peuvent être de tous les types connus par SAMP. Elles peuvent être requises en entrée du comportement et peuvent être accessibles en sortie du comportement ;
2. Ajouter à une vue des propriétés requises par les agents pour qu'ils exécutent le comportement associé. Ces propriétés peuvent être de tous les types connus par SAMP ;
3. Pour les vues de *comportement* et d'*état*, indiquer si durant l'exécution de ces comportements les agents se déplacent dans leur environnement. SAMP automatise le déplacement des agents exécutant un comportement paramétré comme un comportement de déplacement. Cela permet de faire se déplacer les agents (en se référant à leur vitesse et leur direction) sans que l'utilisateur n'ait besoin de modéliser se déplacement ;
4. Pour les vues associées à une vue *état*, indiquer s'il s'agit d'un état *actif* ;

La figure 4.6 montre la fenêtre de SAMP-E permettant de réaliser ce paramétrage. Les chiffres 1 à 4 permettent d'identifier le rôle des zones de paramétrage définies dans l'énumération ci-dessus. Nous voyons dans la suite de cette section un exemple de paramétrage.

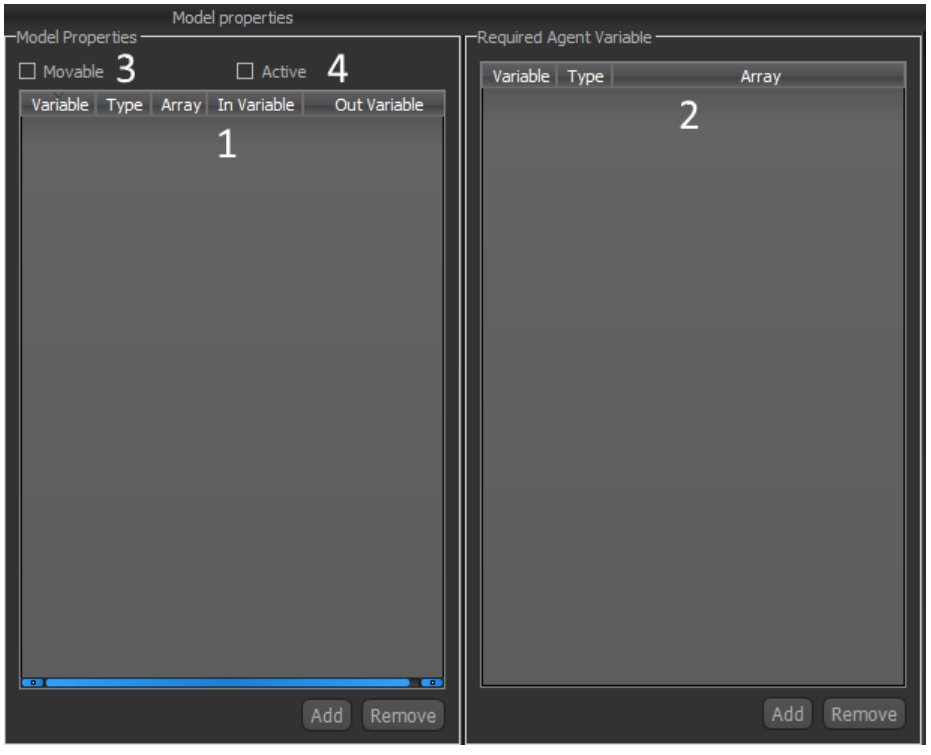

FIGURE 4.6 – Fenêtre de paramétrage d'un modèle

4.6.2/ GÉNÉRALITÉS SUR LES NŒUDS

Nous nous attardons ici pour décrire les généralités concernant les nœuds présents dans les vues de SAMP afin d'avoir toutes les informations nécessaires à la bonne compréhension de la suite de cette section.

Les nœuds des vues de SAMP peuvent posséder deux types différents d'entrées-sorties :

— Des entrées-sorties de types valeurs qui permettent de transmettre des valeurs entre les nœuds d'un graphe de nœuds. Dans SAMP-E les entrées-sorties de valeurs peuvent prendre deux apparences : une apparence ronde ◯ pour la valeur simple et une apparence carrée □ pour les valeurs sous forme de tableau. Chaque type de valeur (entier, flottant, booléen, instance de classe, etc) est représenté par une couleur particulière.

— Des entrées-sorties de types *activation*. Ce type d'entrée-sortie permet d'indiquer aux nœuds du graphe quand ils doivent démarrer leur exécution. Lorsqu'un nœud a terminé de s'exécuter, il active l'une de ses sorties *activation*. Le nœud possédant l'entrée *activation* reliée à la sortie *activation* qui vient de s'activer, s'active alors. Le nœud ayant fini son exécution se désactive. Nous avons appelé cet enchaînement d'activation le *flux d'exécution*. Les entrées-sorties de type *activation* ont une apparence triangulaire ▷ dans SAMP-E, la pointe indique le flux d'activation (en entrée ou en sortie).

La figure 4.7 représente deux nœuds dans SAMP-E. Le premier, à gauche, permettant de créer une entité 3D dans le moteur Shine Engine qui possède une entrée et une sortie d'*activation* ainsi que 8 entrées et une sortie de *valeur* normales. Le deuxième nœud, à droite, permet de récupérer tous les agents présents dans l'environnement. Il possède une entrée et une sortie d'*activation*, deux entrées de *valeur* normales et une sortie de *valeur* sous forme d'un tableau. Une dernière chose à noter, c'est la valeur d'entrée *bShow* du nœud de gauche. Cette valeur d'entrée est de type booléen et possède un bouton à cocher (la forme carrée sous *bShow*) permettant de paramétrer la valeur *bShow* sans être obligé de connecter l'entrée de *valeur* à un autre nœud. Chaque entrée-sortie, lorsqu'elle est connectée à une autre entrée-sortie, voit son apparence être modifiée : la forme qui la représente se remplit (c'est le cas pour l'entrée de valeur *Asker* et la sortie d'activation du nœud `CShEnvironment::GetAllBehaviorAgent`).

Les sorties de valeurs peuvent être de deux types différents : de type valeur de retour ou de type argument. Les sorties de valeurs de type retour correspondent à des sorties correspondant à la valeur de retour de la fonction modéliser par les nœuds auxquels elles appartiennent. Les sorties de type argument correspondent à des arguments passés en paramètre de la fonction des nœuds auxquels elles appartiennent et qui sont des arguments de sorties. Afin de pouvoir les différencier, SAMP-E affiche, au survol de chaque sortie par la souris, une info-bulle indiquant le type de sortie dont il s'agit.

Les nœuds permettant de modéliser les comportements peuvent être de 6 types différents :

1. **Entry** : Ils indiquent les points d'entrée des modèles.
2. **Exit** : Ils indiquent les points de sortie des modèles.
3. **State** : Lorsque le flux de contrôle active un nœud *state*, cela indique que l'agent va changer son état courant pour l'état indiqué par le nœud. Les nœuds *states*

FIGURE 4.7 – Exemple des différentes formes que peuvent prendre les entrées-sorties des nœuds de SAMP

permettent d'exécuter le comportement modélisé dans les vues *états* que nous abordons dans la suite de cette section (cf 4.6.5) ;

4. **Event** : Lorsqu'un *event* connecté à un *state* en cours d'exécution est déclenché, l'agent change d'état pour passer dans celui qui a son entrée connectée à l'*event*. Les nœuds *events* permettent d'exécuter les comportements modélisés dans les vues *événements* que nous abordons dans la suite de cette section (cf 4.6.6) ;
5. **Function** : Ils permettent d'exécuter des fonctions accessibles depuis le moteur Shine Engine ou dans des plugins. Les fonctions accessibles peuvent être des méthodes, des surcharges d'opérateurs ou des blocs de condition ;
6. **Variable** : Ces nœuds permettent de stocker et utiliser des variables présentes dans les modèles. Il peut s'agir de variables possédées par le modèle, de variables globales (provenant de l'environnement par exemple) ou de variable spécifique (delta time par exemple).

Nous avons essayé de trouver un équilibre entre la facilité d'utilisation et une grande expressivité. Le nombre réduit de types de nœuds permet de ne pas alourdir la complexité du système, mais nous estimons que ces différents nœuds offrent une grande expressivité. Nous utilisons ce système dans des développements interne à Shine Research et pour le moment, le système permet de modéliser tous les comportements voulus.

De plus, Shine Engine est un moteur multi-plateforme qui redéfinit les bibliothèques standard, de mathématiques et de système de fichiers. SAMP ayant accès aux fonctions de Shine Engine, cela offre aux utilisateurs une grande panoplie de fonctions prête à l'emploi.

Il est aussi possible d'importer des fonctions (qui seront accessibles sous la forme de nœuds *fonctions*) depuis des plugins. Tout ce que SAMP ne permet pas de faire peut être créé par un utilisateur confirmé (sous forme de code textuel) et importé dans SAMP.

Nous avons ajouté dans SAMP, en plus des nœuds décrits précédemment, des nœuds particuliers afin de faciliter la modélisation et la lecture des vues. Par exemple, nous

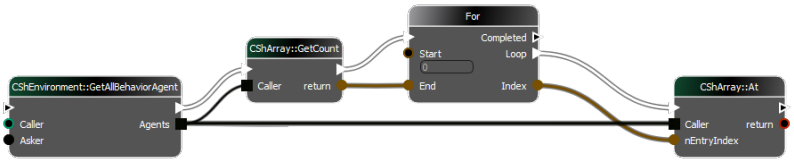

FIGURE 4.8 – Example de la modélisation d'une boucle *for* parcourant un tableau.

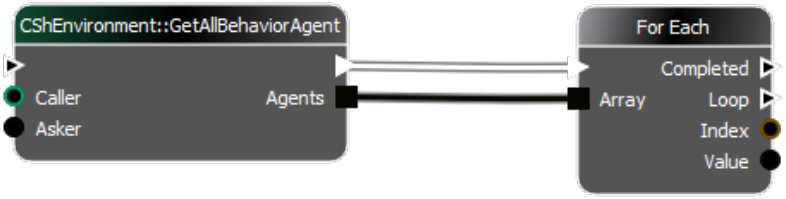

FIGURE 4.9 – Example de la modélisation d'une boucle *foreach* parcourant un tableau.

avons créé un nœud *ForEach* permettant le parcours d'un ensemble et un nœud *IfNull* testant la valeur d'un pointeur. La possibilité d'ajouter des nœuds comme ceux-ci apporte une utilisation simplifiée et permet aux utilisateurs de plus facilement développer des algorithmes complexes et de faciliter la lecture d'une vue. Les figures 4.8 et 4.9 modélisent le même comportement. Dans la figure 4.8, on récupère l'élément courant dans la boucle en sortie du nœud `CShArray::At` alors que dans la figure 4.9 c'est le nœud *foreach* qui fournit cette valeur. Le code généré est exactement le même, mais la lecture et l'utilisation sont simplifiées.

Les nœuds *state* et *event* possèdent une synergie particulière : les sorties *activation* d'un nœud *state* peuvent être de deux types différents : externe ou interne. Lorsqu'une sortie *activation* d'un nœud *state* est externe, elle doit être connectée (directement ou indirectement) à un nœud *event* avant d'être connectée à un nœud state. S'il s'agit d'une sortie *activation* interne, cette sortie doit être connectée (directement ou indirectement) à un nœud *state* sans qu'il y ait de nœud *event* entre les deux nœuds *states*.

Chaque nœud *state* peut posséder une infinité de sorties *activation* de type externe, quelle que soit la modélisation de l'*état* qu'il représente. Les sorties *activation* internes sont quant à elles définies par la modélisation de l'*état*. Nous décrivons comment définir un événement interne dans la section 4.6.5.

Dans la suite de cette section, pour chaque type de modèle, nous donnons des précisions sur le comportement de chaque type de nœud si nécessaire.

La figure 4.10 présente le méta-modèle de SAMP. On constate que chaque nœud peut posséder un nombre indéfini d'entrées-sorties de chaque type. Il existe cependant quelques exceptions :

- le nœud *Variable* ne possède pas d'entrée ou de sortie de type activation ;
- le nœud *Entry* ne possède qu'une sortie de type activation ;
- le nœud *Exit* ne possède qu'une entrée de type activation et une entrée de type valeur ;

On constate aussi sur la figure 4.10 que les liaisons entre les différents nœuds sont contraintes. Ces contraintes sont présentées dans le chapitre 6. Ainsi, il n'est pas possible d'avoir plusieurs connexions sur une même entrée de type valeur. Il n'est pas non plus possible d'avoir plusieurs liens partant d'une même sortie d'activation. De plus, il est

à noter qu'un nœud événement doit, en entrée, être connecté (directement ou indirectement) à un nœud état et en sortie, il doit être connecté (directement ou indirectement) à un nœud état ou un nœud de sortie.

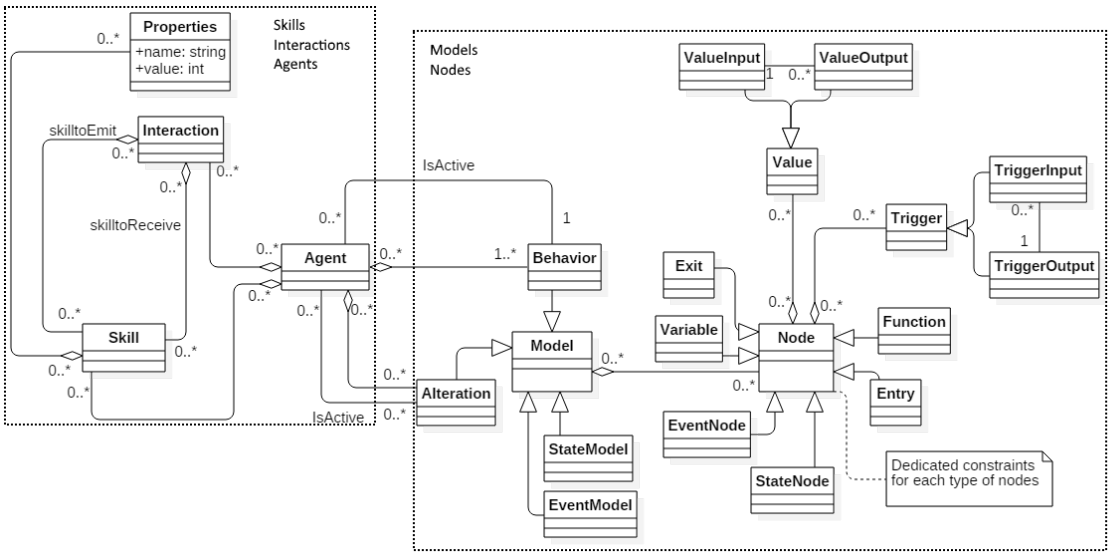

FIGURE 4.10 – Diagramme de classes des éléments composants SAMP.

Pour terminer sur les généralités des graphes de nœuds, nous avons expliqué que les nœuds *state* et *event* permettent, lorsqu'ils sont exécutés, de réaliser le comportement modélisé dans les vues *état* et *événement* correspondantes. Ces nœuds *state* et *event* doivent donc posséder des entrées-sorties correspondants aux entrées-sorties paramétrées pour les vues correspondantes.

Nous l'avons vue dans la sous-section 4.6.1, il est possible d'ajouter aux vues des variables. Ces variables peuvent être des variables d'entrées ou de sortie. Dans la figure 4.11, nous avons la fenêtre de paramétrage de la vue *état LookAt* et un nœud *state Lookat* correspondant à cette vue. La vue *LookAt* recherche, dans une zone définie sur un plan par *xMin*, *yMin*, *xMax* et *yMax*, un agent de type *TargetType*. L'agent exécutant cette recherche se déplace à une vitesse de *speed*. Une fois une cible trouvée, elle est enregistrée dans *Target*. Nous constatons que la vue possède 6 variables d'entrées (*yMin*, *yMax*, *xMin*, *xMax*, *TargetType* et *Speed*) ainsi qu'une variable de sortie (*Target*). Nous retrouvons chacune de ces entrées-sorties sur le nœud correspondant.

Lorsque le nœud est activé, les valeurs d'entrées du nœud sont transmises à la vue et il est possible de récupérer les valeurs de sorties de la vue depuis le nœud.

4.6.3/ VUE ALTÉRATION

Nous l'avons vu, les comportements de SAMP sont divisés en deux types distincts : les réactions et les altérations. Comme nous l'avons expliqué, une vue altération permet de modéliser le comportement de ce qui impacte un agent sans le faire changer directement d'état.

Le premier élément que l'on peut extraire de cette définition, c'est qu'une vue altération ne permet pas l'utilisation de nœuds *state* ni de nœuds *event*. Une vue altération ne

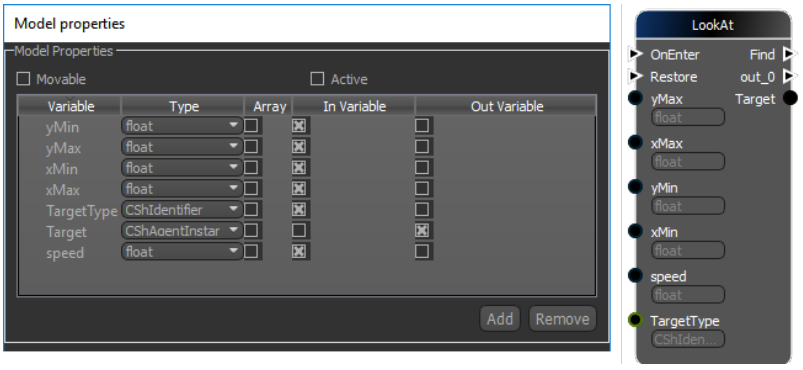

FIGURE 4.11 – Exemple d'un nœud dont les entrées-sorties ont été générées

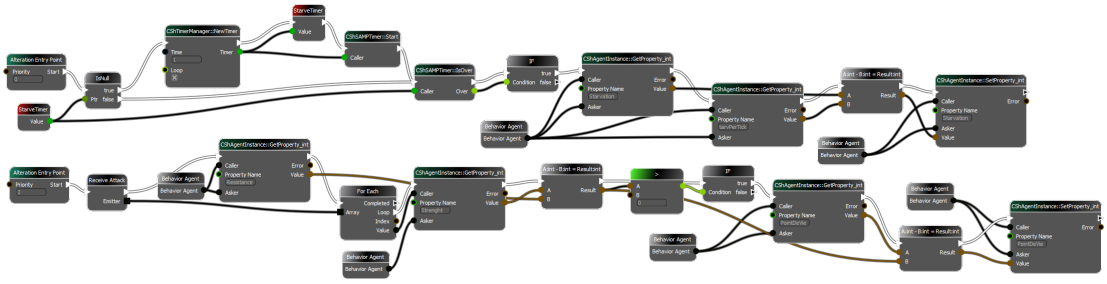

FIGURE 4.12 – Vue altération appliquée aux agents *mouton*

permet pas non plus l'utilisation des nœuds *exit*, car les comportements d'altération sont actifs du début à la fin de la vie d'un agent.

Le fonctionnement de la vue altération est simple : il peut y avoir une infinité de flux d'activation. Ils sont tous exécutés lors de la phase de *PostUpdate*. Le début de chaque flux d'activation est modélisé par un nœud *entry*. Chacun de ces nœuds possède une valeur entière permettant d'indiquer l'ordre dans lequel chaque flux d'activation est activé. A chaque frame, lors de la phase *PostUpdate*, chaque nœud *entry* est activé et le flux d'activation est exécuté et se termine lorsqu'un nœud terminant son exécution n'a aucune de ses sorties d'activation connectées à aucun nœud.

La figure 4.12 représente la vue d'altération des moutons. Dans cette vue *altération* il y a deux flux d'activation :

— Le premier, en haut, permet de gérer la faim des agents moutons. A chaque frame, si le *timer StarveTimer* est terminé, l'agent perd de la faim.

— Le deuxième, en bas, permet de gérer la perte de point de vie en fonction de la résistance de l'agent mouton et de la force de l'attaque qu'il subit (dans le cas où cette fonctionnalité aurait été intégrée à notre exemple fil rouge).

4.6.4/ VUE COMPORTEMENT

Les comportements des agents de SAMP sont divisés en deux catégories : les altérations et les réactions. Dans la partie précédente, nous venons d'aborder les altérations dans leurs fonctionnements et leurs modélisations à travers la vue altération. Nous allons maintenant présenter le fonctionnement du comportement réaction qui lui est modélisé

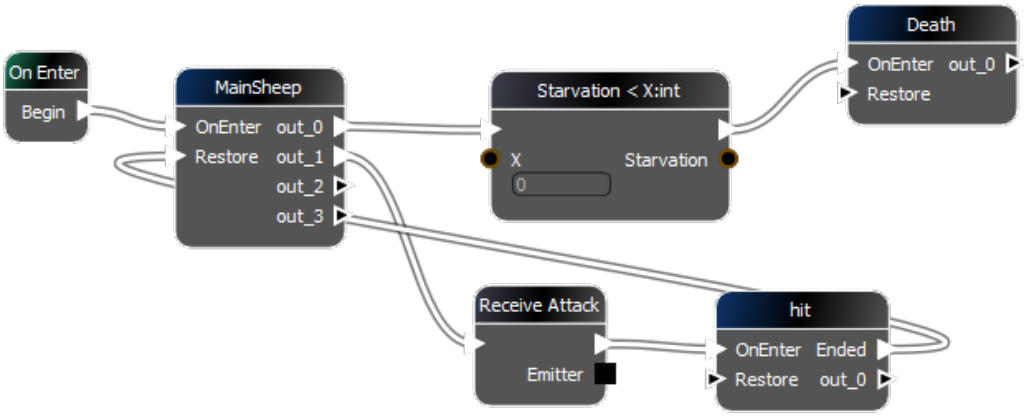

FIGURE 4.13 – Vue comportement modélisant le comportement des agents *mouton*

au travers de 3 vues différentes. La première de ces vues est la vue *comportement*. Cette vue est le point d'entrée du comportement réaction.

Dans la section 4.4.3, nous avons présenté le paramétrage d'un comportement sur un agent. Il permet d'indiquer l'identifiant d'une vue *comportement* associé à l'agent. Un modèle *comportement* possède un seul nœud *entry* permettant de définir le point de démarrage de ce comportement. Ce nœud *entry* doit être directement où indirectement connecté à un nœud *state* ou un nœud *exit* sans qu'un nœud *event* soit activé avant.

Lors de la phase d'initialisation du comportement, le flux d'activation démarre depuis un nœud *entry* et va jusqu'à un nœud *state* ou *exit*. L'initialisation se termine et l'agent a comme état actif le nœud *state* rencontré ou est détruit s'il rencontre un nœud *exit*.

Lorsqu'un nœud *exit* est activé, l'agent est considéré comme mort et il est supprimé de la simulation. Cette mort est définitive. Il ne faut pas confondre cet état de mort à un état de mort d'un personnage dans un jeu qui existerait encore dans le jeu en tant que cadavre.

La vue comportement peut contenir tous les nœuds proposés par SAMP. Il n'y a qu'une seule restriction : une vue *comportement* doit posséder un et un seul nœud *entry*. Si un agent ne possède pas de nœud *exit* c'est qu'il ne sera jamais détruit durant le cycle de vie du SMA dans lequel il se trouve. Il peut mourir et disparaître visuellement, mais ne sera pas détruit.

La vue *comportement* est la vue modélisant le comportement le plus abstrait d'un agent.

La figure 4.13 contient la modélisation du comportement des agents *mouton* de notre exemple dans SAMP-E. On observe que ce comportement est composé de 3 états :

1. Un état *MainSheep* qui permet de factoriser le comportement principal du mouton qui est de se déplacer, se nourrir et se reproduire. Cet état est le premier état activé lorsque l'agent entre dans le comportement décrit ;
2. Un état *hit* activé lorsque l'agent est ciblé par une interaction d'attaque. Dans cet état, l'agent joue une animation signifiant qu'il est attaqué. Lorsque l'animation est terminée, l'agent retourne dans l'état *MainSheep* ;
3. Un état *Death* activé lorsque la valeur de faim de l'agent est inférieure à 0 ;

Nous pouvons constater qu'il est très simple de comprendre les actions exécutées par ce comportement. Il est aussi très simple de modifier les actions réalisées par ce comportement. Par exemple, si l'on désire que les agents *mouton* meurent lorsqu'ils sont

attaqués, il suffirait de relier la sortie d'*exécution* du nœud état *hit* à l'entrée *OnEnter* de l'état *Death*.

La vue comportement est la vue la plus abstraite et permet de modéliser les comportements de haut niveau des agents.

4.6.5/ VUE ÉTAT

Le comportement des agents de SAMP est basé sur le principe décrit dans la section 4.5.1 : les agents exécutent en boucle le comportement défini par un état jusqu'à ce qu'un événement le fasse quitter cet état pour entrer dans un nouvel état (ou pour mourir).

Une vue *état* permet de définir chaque comportement que les agents vont exécuter lorsqu'ils seront dans cet état.

Une vue état permet de définir et gérer 4 flux d'activation différents. Chacun de ces flux correspond à une phase de fonctionnement de la vue :

1. Un flux d'activation lorsqu'un agent entre dans l'état. Il s'agit en quelque sorte de la phase d'initialisation de l'état et elle est appelée *OnEnter* ;
2. Un flux d'activation lors de la phase de *PreUpdate* du système. C'est durant cette phase que l'agent exécutera les actions définies par l'état ;
3. Un flux d'activation lors de la phase de *PostUpdate* du système. C'est dans cette phase que les tests pour des événements internes (décrits dans la suite de cette section) seront exécutés ;
4. Un flux d'activation lorsqu'un agent quitte l'état. Ce flux est appelé *OnLeave*.

Aucune de ces phases n'est obligatoire et il est envisageable d'avoir un état qui ne définit aucun comportement (pour un état de mort par exemple). Pour indiquer le point de départ de chacun de ces flux d'activation une vue état possède 4 types de nœud *entry*, chacun correspondant à un flux d'activation.

Nous l'avons vu précédemment, pour quitter un état, il est nécessaire qu'un événement lié à cet état soit déclenché. Nous avons aussi vu que des nœuds événements peuvent être connectés aux sorties *activation* d'un nœud *state*. Dans ce cas, nous appelons ces événements des événements externes.

La figure 4.13 montre la modélisation dans SAMP-E de la vue *comportement* des agents *mouton*. Dans cette vue, nous avons un nœud *state MainSheep* dont deux sorties d'*activation* sont reliées à des nœuds *event* : `Starvation < X:int` et *Receive Attack*. Il s'agit ici d'événements externes qui ne dépendent pas du résultat de l'exécution de l'état *MainSheep*.

Nous avons aussi abordé les événements internes d'un état. Il s'agit d'événements dont les conditions d'activation sont modélisées dans le flux d'activation de la phase *PostUpdate* de la vue *état*. Dans la vue *état*, pour indiquer qu'un événement interne est déclenché, il faut activer un nœud *exit*. Ainsi, lors de l'exécution du comportement d'un état, si un nœud *exit* est activé, cela déclenchera un événement interne à l'état. Les événements internes permettent, par exemple, aux états possédant des objectifs internes (se déplacer vers une position, rechercher un agent dans l'environnement, ...) ou un temps d'exécution (durée d'une animation, durée d'un *chronomètre*, ...) d'indiquer qu'ils ont terminé leur exécution.

La figure 4.14 montre la modélisation de la phase *PostUpdate* de l'état *MoveTo* dans SAMP-E. Cet état a pour but de faire se déplacer un agent vers une destination. Lorsque la destination est atteinte, l'état déclenche un événement internet permettant d'indiquer à l'agent qui exécute le comportement de cet état qu'il est arrivé à destination et qu'il doit quitter l'état *MoveTo*. Durant cette phase de *PostUpdate*, le flux d'activation va activer le nœud *exit* lorsque la destination (donnée par la valeur du nœud *Target*) est atteinte par l'agent. Le fait d'activer ce nœud *exit* déclenche un événement qui indique à l'agent qu'il faut quitter l'état *MoveTo*.

Dans la figure 4.15, le nœud *MoveTo* est une instance de l'état *MoveTo*. Lorsque ce nœud est activé l'agent entre dans l'état *MoveTo* est exécute son comportement. Lorsque l'agent a atteint sa destination, l'événement interne de l'état *MoveTo* est déclenché ce qui active la sortie d'*activation PositionReached* du nœud *MoveTo*.

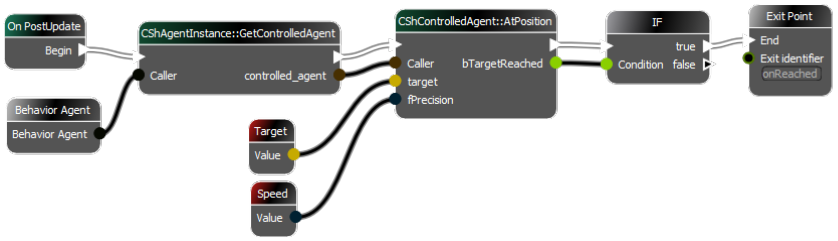

FIGURE 4.14 – Modélisation de la phase *PostUpdate* de l'état *MoveTo*

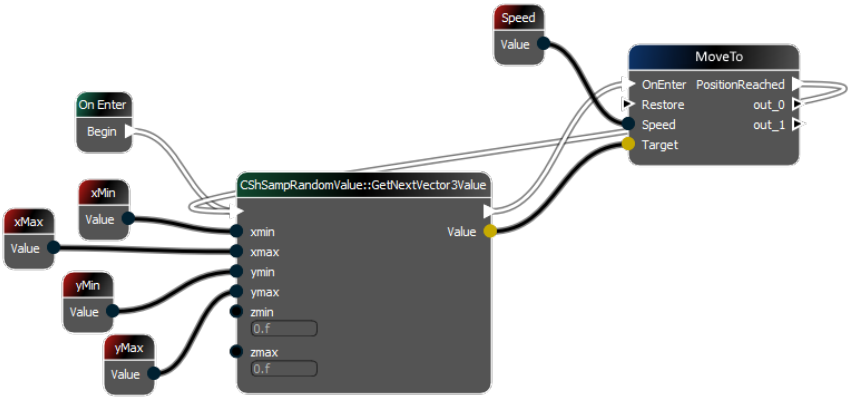

FIGURE 4.15 – Modélisation de l'état *random move*

La figure 4.16 présente la fenêtre de paramétrage de l'*état MoveTo*. On constate tout d'abord, en haut de la fenêtre, que l'état est *movable* et *active*. Le fait que l'état soit *movable* indique que l'agent pourra se déplacer quand il sera dans cet état. Ce qui permet d'automatiser le déplacement de l'agent en ne paramétrant que sa vitesse et sa direction. Le fait que l'état soit *active* indique que les agents ayant pour état courant cet état, seront actifs dans la gestion de leurs interactions et exécuteront un scan de leur environnement (cf section 5.2).

La partie centrale de la fenêtre recense les variables présentes dans l'état. Ces variables possèdent un identifiant, un type et trois valeurs booléennes déterminant si les variables sont des tableaux (array), sont passées en entrées de l'état (`In Variable`) ou récupérées en sorties de l'état (Out Variable).

Dans l'état *Déplacement*, il est nécessaire d'avoir 4 variables :

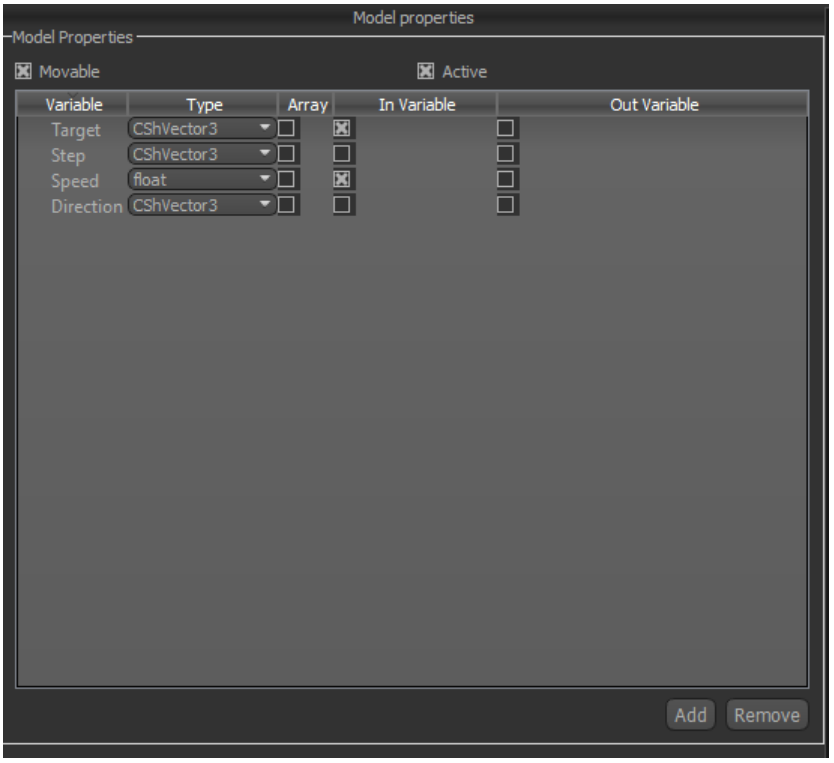

FIGURE 4.16 – Fenêtre de paramétrage du modèle *état Déplacement*

— *Target* contenant la destination à atteindre est passée en entrée de l'état

— *Step* qui est une variable paramétrée à l'activation de l'état. Elle contient le *pas* de déplacement effectué pour chaque milliseconde.

— *Speed* qui est une variable passée en entrée de l'état. Elle contient la vitesse de déplacement de l'agent.

— *Direction* qui est une variable indiquant la direction du déplacement de l'agent. Cette variable est paramétrée lors de l'activation de l'état en fonction de la *Target* et de la position actuelle de l'agent.

La figure 4.15 représente une vue *état* dans laquelle est modélisé, dans SAMP-E, un comportement de déplacement aléatoire. Dans cette figure, nous voyons un nœud *function* (*GetNextVector3Value*) permettant de générer, lors de la phase *OnEnter* de l'*état*, une valeur aléatoire de type *Vector3*. Ce nœud *function* est activé lors de l'activation de l'état par le biais du nœud *entry* de type *On Enter*. Le nœud *function* prend en entrée 6 valeurs permettant d'indiquer les valeurs minimums et maximums du *Vector3* généré. Dans notre exemple, même si nous utilisons la 3D, toutes les positions sont situées sur un plan. C'est pourquoi les valeurs minimum et maximum en Z sont fixées à 0. Les autres valeurs (pour X et Y) proviennent de valeurs passées en entrée de l'état et stockées dans les variables *xMin*, *xMax*, *yMin* et *yMax*.

Lorsque la fonction a généré le *Vector3* elle active le nœud *state StateMove*. Ce nœud prend en entrée deux valeurs : une valeur de vitesse initialisée avec une valeur passée en entrée de l'état *Déplacement aléatoire* (*Speed*) et la destination du déplacement initialisée avec la valeur générée par la fonction précédemment décrite. Lorsque l'événement interne *Position Reached* de l'état *StateMove* est déclenché, la *fonction* de génération

est à nouveau activée pour recommencer une boucle de génération de destination et de déplacement.

Le nœud *StateMove* est un nœud *state* qui, lorsqu'il est activé, va exécuter le comportement défini par l'état *StateMove*. Cet état *StateMove* est modélisé dans les figures 4.17, 4.18 et 4.14.

Cet état exécute un déplacement vers une destination donnée en tant que paramètre d'entrée. L'*état MoveTo* est découpé en trois parties, chacune modélisant le comportement d'une phase : la phase *OnEnter* (fig. 4.17), *PreUpdate* (fig. 4.18) et *PostUpdate* (fig. 4.14). Durant la phase *OnEnter*, l'*état* calcule la direction du déplacement en fonction de la position de l'agent exécutant le comportement et la destination ciblée. Ensuite, un *ControlledAgent* est initialisé avec les valeurs de direction et de vitesse du déplacement. Le *ControlledAgent* est un objet spécifique à SAMP permettant de contrôler facilement le déplacement des agents de SAMP et le positionnement de leur représentation graphique dans Shine Engine. Lorsqu'un comportement est paramétré comme *Movable* (c'est-à-dire que les agents exécutant ce comportement peuvent se déplacer) une instance de *Controlled Agent* est automatiquement créée et utilisable par le comportement. Durant la phase de *PreUpdate*, le *ControlledAgent* est mis à jour. Enfin, dans la phase de *PostUpdate*, l'*état* teste si la destination est atteinte. Si c'est le cas, il active un nœud *exit* déclenchant un événement interne permettant d'indiquer que l'*état* a terminé son exécution.

Dans ces figures, on constate la présence de nœuds *Behavior Agent*. Ces nœuds permettent d'accéder, depuis une vue, à l'instance de l'agent exécutant le comportement modélisé. Le nœud *Delta Time* permet d'obtenir, durant les phases *PreUpdate* et *PostUpdate*, le temps écoulé depuis la dernière frame (cf section 2.1.3).

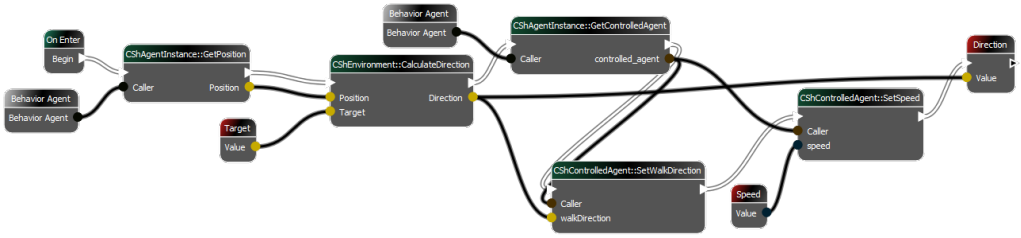

FIGURE 4.17 – Modélisation de la phase *OnEnter* de l'état *MoveTo*

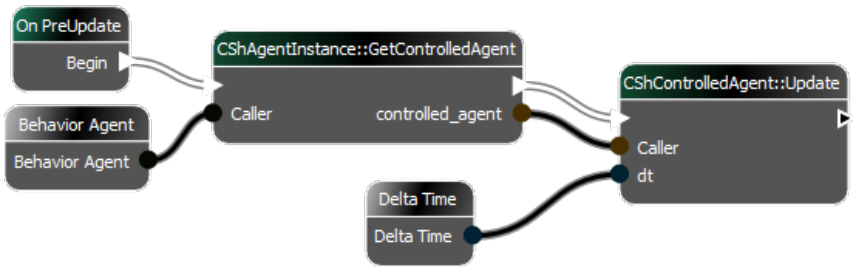

FIGURE 4.18 – Modélisation de la phase *PreUpdate* de l'état *MoveTo*

4.6.6/ VUE ÉVÉNEMENT

Le dernier type de modèle que nous avons mis en place dans SAMP est le type *événement*. Ce type de vue permet de modéliser les conditions de déclenchement des événements permettant aux agents de SAMP de changer d'état.

Ces vues ne possèdent qu'un seul flux d'activation (et donc qu'un seul nœud *entry*).

Lorsque l'entrée d'activation d'un nœud *event* est connectée à une sortie d'activation d'un nœud *state*, à chaque frame, l'événement correspondant au nœud *event* est testé. Lors de ce test, le flux d'activation de l'événement démarre du nœud *entry* de cet événement. Lorsqu'un nœud *exit* est activé, cela signifie que les conditions de déclenchement de l'événement sont réunies et l'événement est déclenché. Si un nœud d'une vue *événement* termine son exécution et que sa sortie d'activation activée n'est reliée à aucun autre nœud cela signifie que l'événement n'est pas déclenché.

Les événements peuvent retourner des valeurs. Pour qu'une valeur puisse être récupérée lorsqu'un événement s'active, il faut que dans la vue de l'événement une propriété de la vue soit paramétrée en tant que propriété de *sortie*. Lorsque l'événement est déclenché, cette propriété sera alors accessible.

Certains événements sont générés automatiquement et ne requièrent pas de créer des vues pour être utilisables sous forme de nœuds. Il y a deux types d'événements dans ce cas là :

— Les événements déclenchés lorsqu'une interaction est reçue par un agent ;

— Les événements déclenchés lorsqu'une comparaison d'une propriété d'un agent est vérifiée.

Dans la figure 4.13, il y a un nœud *Receive Attack* qui est un nœud événement activé lorsque l'agent a reçu une interaction signifiant qu'il est attaqué. La valeur de sortie de ce nœud événement contient la liste de toutes les interactions d'attaque reçues durant la frame courante. Dans cette figure, il y a aussi un nœud `Starvation < X:int` qui est un événement déclenché lorsque la propriété *Starvation* de l'agent est inférieure à la valeur indiquée par *X*. La valeur de retour de ce nœud contient la valeur de la propriété *Starvation* au moment du déclenchement de l'événement.

Les vues *événement* ne peuvent contenir de nœud *event* ou de *state* ce qui signifie que leurs flux d'activation sont parcourus entièrement durant la frame où ils sont activés.

La figure 4.19 expose la vue modélisant l'*événement ISNear* de notre exemple. Cet événement se déclenche lorsqu'un agent exécutant le comportement est proche d'une position donnée en paramètre d'entrée de l'événement. Cet événement prend en entrée la position cible et la distance à partir de laquelle l'événement considère l'agent comme proche de sa cible.

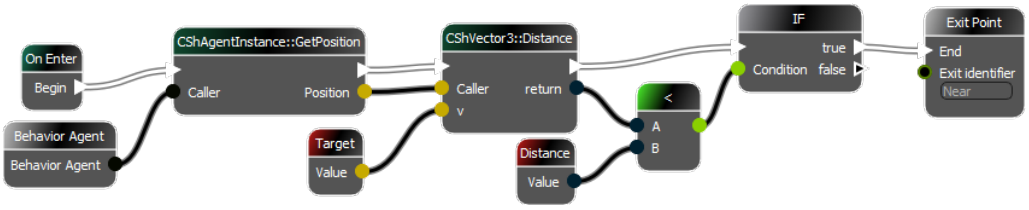

FIGURE 4.19 – Modélisation de l'événement *IsNear*

4.6.7/ INSTANCIATIONS DES AGENTS

Lorsque tous les comportements ont été modélisés, que les interactions, les compétences et les ressources *agents* ont été paramétrées, il reste une dernière étape avant d'exécuter le système à l'aide de SAMP-X. Cette étape consiste à paramétrer les instances d'agent qui seront présentes dans la simulation.

Dans la section sur les agents (4.4), nous avons décrit les agents en tant que ressources. Dans cette section, nous abordons les instances de ces ressources qui seront les agents présents durant l'exécution du système.

Il existe deux solutions pour instancier des agents dans SAMP-X :

1. Utiliser un nœud *function* spécial permettant la création d'un nouvel agent. Ce nœud prend en entrée le type du nouvel agent, le comportement qu'il exécutera, sa position, la ressource permettant sa représentation graphique et une liste de paramètres correspondant aux propriétés de l'agent. Cette liste permet d'initialiser les propriétés de l'agent lors de sa création.
2. Lors du chargement d'une scène (un niveau de jeu), SAMP-X parcours tous les objets de la scène chargée et créé un agent pour chaque objet de la scène possédant une *dataset*[2] *agent*. La figure 4.20 montre une scène représentant notre exemple. Dans cette scène, chaque objet est paramétré pour qu'un agent soit créé lors du changement de la scène : chaque loup, chaque mouton et chaque carré d'herbe. On voit, en bas à gauche de la figure, la zone de propriétés de l'objet sélectionné (dont un zoom est affiché sur la droite de la figure). On voit qu'il est un agent de type *Sheep*. On voit aussi qu'il est possible de donner des valeurs aux propriétés de l'agent. Ces propriétés correspondent aux propriétés paramétrées sur les ressources agents (section 4.4). Dans la figure 4.20, l'objet sélectionné est une entité 3D représentant un loup. Elle est paramétrée pour instancier un agent *Wolf* et initialiser les différentes propriétés de cet agent.

SAMP permet donc, à l'aide de Shine Engine et de son éditeur, de modéliser les scènes 2D ou 3D des SMA exécutés dans SAMP-X et de paramétrer les instances des agents présents au démarrage de la scène.

TABLE 4.6 – Instances des agents de la simulation

Nom	Nb	Type	Cpt	Propriétés	Capacités
A	1	Mouton	—	—	—
B	1	Mouton	—	Résistance = 4	—
C, D	2	Loup	—	—	—
E	1	Loup	—	PdV = 9	—

La table 4.6 expose un exemple de différentes instances d'agents correspondant à notre exemple de proies-prédateurs. Chaque instance d'un agent doit être d'un type défini lors de la modélisation de la population (table 3.1). Dans la table 4.6, nous paramétrons l'instanciation de 5 agents, 2 de type mouton et 3 de type loup. Nous voyons que parmi les agents de type loup, l'agent E modifie la valeur par défaut du nombre de points de vie

2. Les *datasets* sont des structures de données que l'on peut ajouter aux propriétés d'un objet afin d'étendre les informations qu'il expose. Pour SAMP, ces *datasets* permettent d'indiquer quels objets représenteront des agents dans SAMP-X et de donner des valeurs aux propriétés des agents qui seront créés.

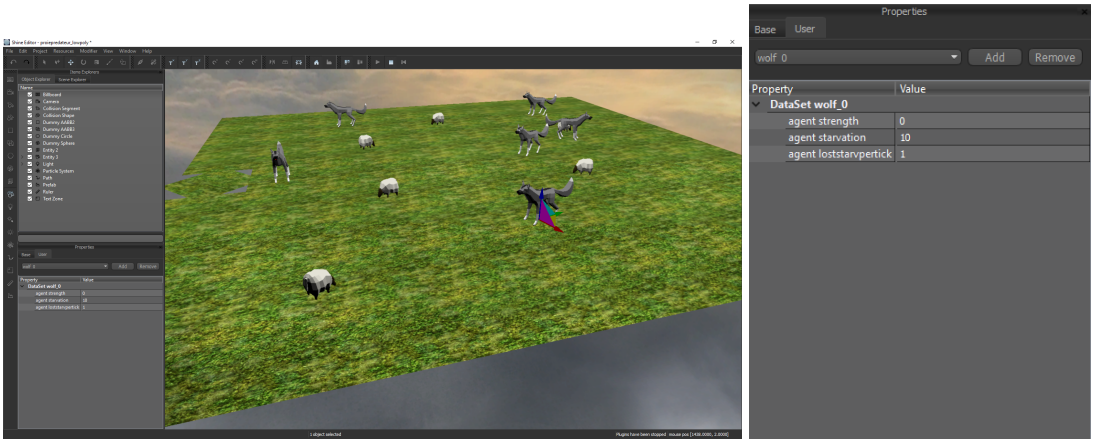

FIGURE 4.20 – Scène modélisée dans Shine Engine.

et que l'agent B modifie la valeur de résistance par défaut des agents *mouton*. Les valeurs par défaut des propriétés des agents ont été paramétrées lorsque l'on a défini les différents types d'agents et que des propriétés leur ont été assignées (cf 4.4).

Il n'y a pas d'agent *herbe* présent dans cette table, car dans l'exemple, les agents *herbe* seront générés aléatoirement au démarrage du système.

4.7/ BILAN

SAMP, aidé de ses trois modules, permet la modélisation et le paramétrage de toutes les composantes d'un système multi-agent. De la création des agents à leur instanciation en passant par la modélisation de leur comportement, SAMP permet aux utilisateurs débutants de créer des SMA sans utiliser de langage textuel, mais offre aussi la possibilité aux utilisateurs confirmés d'importer du contenu dans SAMP.

Le méta-modèle SAMP-M a été réduit au maximum afin d'offrir une grande expressivité en conservant une grande accessibilité. Enfin, l'instanciation des agents, rendue possible par l'utilisation de l'éditeur de Shine Engine, permet une plus grande compréhension de ce qui est modélisé et qui sera exécuté.

SAMP est utilisé pour modéliser notre exemple fil rouge. Nous l'avons aussi utilisé pour modéliser un autre système qui est un jeu vidéo de plateforme en 2D que nous abordons dans la section 8.6. Certains éléments sont encore perfectibles. Notamment la compréhension de certains graphes de nœuds lorsque les vues modélisées contiennent beaucoup de nœuds. Certains paramètres peuvent paraître obscurs et un soin particulier sera apporté à la documentation et à l'affichage d'info-bulles à destination des utilisateurs.

Les surcoûts potentiels dus à l'implémentation des compétences et des interactions sont maîtrisés. Ce système d'automatisation apporte un véritable avantage tant au niveau de la simplicité de modélisation, mais aussi du fait qu'il permet de gérer une partie des règles du jeu.

L'approche états-événements apporte une optimisation des calculs en évitant de gérer des événements qui n'ont pas d'intérêt en fonction des états dans lesquels se trouvent les agents. De plus, cette approche permet elle aussi de paramétrer une partie des règles

du jeu en empêchant les agents de réaliser des actions qui ne leur sont pas permises en fonction des états dans lesquels ils se trouvent.

L'apport le plus important est sur la facilité d'utilisation et la simplicité de modélisation. La section suivante se concentre sur le fonctionnement et l'implémentation des interactions afin d'étudier s'il y a la possibilité d'améliorer leur traitement.

5

LES INTERACTIONS

Les jeux vidéo mettent en scène des univers où se côtoient de nombreux personnages et entités. Chaque personnage ou entité est capable d'interagir avec les autres. Ces interactions permettent de rendre l'univers "vivant", cohérent et immersif. Dans certains jeux vidéo les interactions sont partie intégrante de la jouabilité. Les interactions sont présentes partout : lorsqu'un personnage en voit un autre, lorsqu'un personnage en touche un autre ou lors d'une communication.

A chaque frame, chaque agent peut potentiellement rechercher des interactions qui pourraient l'intéresser et qui auraient été laissées dans l'environnement par d'autres agents. Ou bien, chaque agent pourrait rechercher quels autres agents sont intéressés par des interactions qu'il va émettre.

Dans les SMA, un agent obtient des informations, à propos de son environnement ou des autres agents, en scannant son environnement proche ou en étant notifié par son environnement ou d'autres agents. Cette capacité à percevoir son environnement est appelée *perception*. Danny Weyns et al. [86] décrivent les perceptions comme "*la capacité d'un agent à ressentir son environnement proche, résultant en une perception de l'environnement*". Quand les perceptions sont utilisées, il y a deux possibilités : la première est basée sur le fait que l'agent émetteur notifie le récepteur. La seconde se base sur le fait que l'agent récepteur va scanner son environnement proche pour découvrir des interactions qui pourraient le concerner.

Andrew Frank [38] explique "*qu'un agent peut être décrit par une fonction de perception et une fonction de décision*". Ruzena Bajcsy [11] quant à lui explique *"qu'il devrait être axiomatique qu'une perception n'est pas passive, mais active. Ainsi, l'activité de percevoir c'est explorer, sonder, chercher. [...] Nous ne faisons pas que voir, nous regardons."*. Frank et Bajcsy sont d'accord pour définir un agent comme actif quand il perçoit son environnement. Dans le développement de notre approche, nous ne sommes pas aussi définitifs.

Nous considérons les perceptions et les communications comme des interactions. Une communication est une interaction où chacun des agents est conscient qu'il échange des informations. Nous avons vu quelques possibilités pour un agent d'envoyer une interaction aux autres : par l'approche **tableau blanc**, par l'approche des *phéromones* [28] ou par l'approche des **notifications** [18].

Dans chacune de ces approches, les agents sont actifs dans le management des perceptions : en écrivant ou en lisant le tableau blanc, en plaçant ou en cherchant des phéromones ou en notifiant les agents récepteurs. Dans le cas des notifications, seul l'agent

émetteur est actif, le récepteur pouvant être inactif (jusqu'à la réception de la notification).

Nous en avons déduit la définition suivante :

Définition (Approche classique)**.** *L'approche classique est un système dans lequel les agents (émetteur et récepteur) doivent être actifs durant leurs perceptions.*

5.1/ L'APPROCHE CLASSIQUE

Afin d'optimiser au mieux les SMA développés à l'aide de SAMP, nous avons cherché à optimiser ce qui nous semblait le plus consommateur de ressources. Dans un SMA, les interactions entre les agents sont très nombreuses et peuvent représenter une part importante de la consommation de ressources. Dans un jeu vidéo, chaque personnage non-joueur (PNJ)[1] est capable de percevoir son environnement, les autres PNJ et le joueur. Dans des jeux vidéo très peuplés, les interactions entre les PNJ se révèlent très nombreuses.

Pour commencer, nous posons cette question : *Si un arbre tombe dans une forêt, mais que personne n'est là pour l'entendre, fait-il du bruit ?*[2]

Cette question métaphysique amène une réflexion intéressante pour les SMA : le fait d'émettre un son est une interaction à l'encontre des agents capables de l'entendre. Mais si personne n'entend ce son, pourquoi consommer des ressources en l'émettant.

Dans un SMA, cet exemple pourrait avoir deux solutions :

1. Tout au long de sa chute, l'arbre va scanner son environnement pour voir si des agents sont capables de percevoir son interaction sonore et va les notifier s'il en trouve ;
2. Au début de sa chute, l'agent va placer dans l'environnement un marqueur indiquant qu'une interaction sonore est en cours. Les agents à proximité devront scanner leur environnement pour pouvoir utiliser cette interaction.

La première solution est très gourmande, car l'agent émetteur de l'interaction va scanner inutilement son environnement à la recherche d'agents cibles qui n'existent pas.

La seconde solution semble plus économe, mais elle possède un autre problème. Prenons un agent que nous plaçons dans une pièce. Cet agent va continuellement scanner son environnement afin d'être notifié si une interaction apparait à proximité.

Dans ces deux cas, nous avons des agents qui consomment inutilement des ressources que ce soit au moment de l'émission d'une interaction ou en scannant l'environnement à la recherche d'interactions.

5.2/ INTERACTIONS INVERSÉES ET AGENTS ACTIFS/PASSIFS

Afin de pallier les problèmes de sur-consommation amenés par le fonctionnement de l'approche classique, nous avons développé une nouvelle approche de gestion des interactions.

1. Dans un jeu vidéo, les personnages non-joueur sont toutes les entités (agents) peuplant l'environnement, mais qui ne sont pas contrôlés par le joueur.
2. https ://en.wikipedia.org/wiki/If_a_tree_falls_in_a_forest

Pour développer cette approche, nous avons isolé les deux principaux problèmes de l'approche classique :

1. Les agents émettent des interactions même si aucune cible n'est présente pour les recevoir ;
2. Les agents scannent leur environnement même si aucune interaction n'est émise dans leur proximité.

Pour pallier aux problèmes qui font que les agents émettent des interactions sans que des agents cibles ne soient à proximité, nous avons développé une approche basée sur des agents actifs et passifs. Nous posons ces deux définitions :

Définition (Agent Actif)**.** *Un agent est actif lorsque les actions qu'il réalise impactent sa perception du système ou que les actions qu'il réalise impactent la perception des autres agents du système.*

Définition (Agent Passif)**.** *Un agent passif est un agent qui n'est pas actif.*

Au cours de son existence, un agent change de statut en fonction de l'état dans lequel il se trouve. Lorsque l'on paramètre une vue *état*, il est possible d'indiquer s'il s'agit d'un état actif ou passif. Lorsqu'un agent entre dans un état, il acquiert le statut paramétré par l'état. Dans notre exemple, les états actifs sont les états de déplacement (*moveTo*). Seuls les agents actifs vont scanner leur environnement. Nous voyons par la suite les autres actions que réalisent les agents actifs.

Cette distinction agents actifs et passifs est complétée par un système de tableaux d'interactions à émettre permettant aux agents de savoir vers quels autres agents émettre quelles interactions. Cette fonctionnalité est proche des approches *publish/subscribe* [39, 31] à ceci près que SAMP, ce ne sont pas obligatoirement les agents intéressés par les interactions qui vont souscrire eux mêmes sur ces interactions. Dans le cas de systèmes *publish/subscribe*, lorsqu'un agent est intéressé par une interaction, c'est lui même qui a la tâche d'aller souscrire à cette interaction.

Chaque tableau contient une liste de couples composés de l'interaction à émettre et de l'agent cible de l'interaction. Chaque agent possède un de ces tableaux et, à chaque frame, le tableau de chaque agent est parcouru. Pour chaque élément du tableau, l'interaction enregistrée est envoyée à l'agent qui lui est associé.

Ces tableaux d'interactions à émettre peuvent être remplis par n'importe quel agent qui est actif. Lorsqu'un agent est actif, il scanne son environnement à la recherche :

1. d'agents qui pourraient être intéressés par des interactions qu'il peut émettre ;
2. d'agents pouvant émettre des interactions qui pourraient l'intéresser.

Dans le premier cas, l'agent va enregistrer, pour chacune des interactions qu'il peut émettre, les agents intéressés par ces interactions. Dans le deuxième cas, il va lui-même s'enregistrer dans les tableaux d'interactions à émettre des agents qu'il a scannés.

Afin d'optimiser cette phase de scanne de l'environnement, nous avons diminué le nombre de scans effectués par certains agents. Lorsqu'un agent *A*, en scannant son environnement, détecte un agent *B*, *A* va vérifier si *B* peut être la cible d'interactions qu'il peut émettre. *A* va aussi vérifier s'il peut être la cible d'interactions émises par *B*. Si dans la même frame, *B* scanne son environnement, il devrait aussi scanner *A*. Comme *A* a déjà scanné *B*, il n'est pas nécessaire que *B* exécute ces vérifications. Ainsi, chaque agent possède un autre tableau contenant une liste d'agents qu'il a déjà scannés ou qui les ont

déjà scanné. Lorsqu'un agent détecte, lors d'une phase de scan de l'environnement, un autre agent, il vérifie d'abord si cet agent l'a déjà scanné en parcourant son tableau. Si c'est le cas, il ne va pas analyser cet agent et va continuer son scan de l'environnement.

Avec notre système, nous espérons réduire le nombre de scans de l'environnement effectués par les agents pour le fonctionnement des interactions. Nous avons vu comment les agents peuvent s'enregistrer dans les tables d'interactions à émettre. Il reste à décrire comment chaque agent peut se désinscrire des listes dans lesquelles il est inscrit lorsque c'est nécessaire. Chaque agent possède un tableau dans lequel on ajoute un élément lorsque l'agent est ciblé par une interaction. Chaque élément est un couple composé de l'agent émetteur et de l'interaction émise. A chaque frame, en plus de vérifier les agents à proximité intéressés par leurs interactions, les agents actifs vont vérifier s'ils doivent désinscrire des agents de leur tableau d'interactions à émettre ou se désinscrire eux-mêmes des tableaux d'interactions à émettre des autres agents.

Il existe alors quatre possibilités, pour un agent, d'être désinscrit d'un tableau d'interaction à émettre.

1. Lorsque l'agent émetteur cesse d'émettre l'interaction ;
2. Un agent émetteur actif désinscrit les agents qui ne sont plus capables de recevoir les interactions qu'il émet (obstacle, distance, perte de compétence ...) ;
3. L'agent récepteur actif se désinscrit des interactions qu'il n'est plus capable de recevoir ;
4. L'agent récepteur meurt.

Toutes ces fonctionnalités sont transparentes pour l'utilisateur. Il n'a pas besoin de savoir comment les interactions sont émises pour pouvoir utiliser ce système. Le but étant d'optimiser les interactions tout en gardant la simplicité d'utilisation de SAMP. Nous allons dans la suite de ce chapitre aborder l'envoi des interactions.

5.3/ L'ENVOI DES INTERACTIONS

Dans les SMA les interactions, qu'il s'agisse de communications verbales ou de perceptions, permettent aux agents d'échanger des informations. Lorsque des agents interagissent entre eux, ils s'échangent des informations.

Afin de rendre ces informations compréhensibles par tous les agents du système, il est nécessaire de les formaliser. Si un agent émet une interaction à destination d'un autre agent, il est nécessaire que l'agent récepteur puisse recevoir ce message, mais aussi qu'il puisse le comprendre.

En plus du fait que les messages doivent être lisibles et compréhensibles pour les agents qui les reçoivent, ils doivent permettre de transmettre des informations diverses et variées. Lorsqu'un agent attaque un autre agent, l'interaction émise doit contenir son type (une attaque) et des informations comme la puissance de l'attaque, le type d'attaque (tranchant, perforant, ...) ou de possibles effets secondaires (empoisonnement, brûlure, renversement, ...).

Les interactions ont toutes le même format. Elles sont composées : du type de l'interaction, de son identifiant, de son destinataire, de son émetteur et d'un tableau contenant

les valeurs utiles à l'analyse de l'interaction. Ce dernier tableau peut contenir des valeurs de n'importe quel type afin de pouvoir transmettre tout ce dont a besoin la cible de l'interaction pour l'analyser.

Dans notre exemple, lorsqu'un agent *loup L* attaque un agent *mouton M*, l'interaction est composée comme suit :

— Type : "*Attaque*" ;

— Identifiant : Attaque_001 ;

— Destinataire : "*M*" ;

— Emetteur : "*L*" ;

— Valeurs : «"Force", 1» ;

Dans cet exemple, l'interaction *attaque* transmet la valeur de force de l'attaquant pour que la cible puisse agir en fonction de cette information. Mais cette même interaction aurait pu être plus complexe et, par exemple, transmettre une information sur le type de l'attaque. Dans ce cas, nous aurions ajouté un élément dans le tableau : <"Type", e_type_tranchant>. La valeur `e_type_tranchant` aurait été de type énumération. Cette information aurait pu être utilisée afin d'infliger plus de dégâts aux moutons qui auraient été sensibles aux attaques tranchantes.

Afin de pouvoir analyser toutes les interactions qui l'ont ciblé durant une même frame, chaque agent possède un tableau, vidé au début de chaque frame, dans lequel il enregistre toutes les interactions reçues lors de la frame courante. A chaque frame, durant la phase de *PostUpdate*, chaque agent va parcourir ce tableau et réagir, ou non, à chaque interaction. Il est possible pour l'agent de réagir de trois manières différentes à une interaction :

1. S'il est dans un état connecté à cette interaction, il va alors changer d'état. Ceci est possible, car dans SAMP, lorsqu'une interaction est déclenchée, elle génère un événement. Et les événements sont les sources de changement d'état ;
2. Il peut, indépendamment de l'état dans lequel il est, être altéré par cette altération et donc modifier ses propriétés en fonction ;
3. Il peut enfin décider de ne pas réagir à cette interaction qui, dans son état actuel, n'a pas d'intérêt pour lui.

Il est à noter que lors d'une même frame, un agent peut être la cible de plusieurs interactions du même type en provenance de différents agents. Dans ce cas-là, si l'agent réagit aux interactions de ce type, il devra analyser les données de chaque interaction de ce type qu'il a reçue durant cette frame.

Dans les vues de SAMP, il existe un nœud fonction particulier permettant de faire l'envoi d'une interaction. La figure 5.1 expose deux nœuds. Le premier, à gauche, permet à un agent d'émettre l'interaction *Attack*. On constate que ce nœud requiert deux valeurs en entrée. La première, *Target* correspond à la cible de l'interaction. La seconde, *Strenght*, correspond à la valeur de force de l'agent émetteur requise par l'interaction.

Le second nœud, à droite, est un nœud permettant aux agents d'être notifiés lorsqu'ils sont ciblés par une interaction *Attack*. Les nœuds de réception d'interactions fonctionnent comme les nœuds *event* : ils doivent être connectés à une sortie d'activation d'un nœud *state* et permettent, lorsqu'ils sont déclenchés, de faire changer d'état un agent. Un nœud

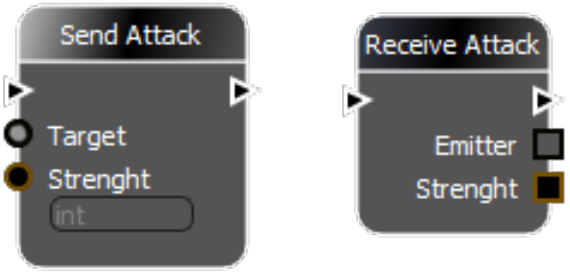

FIGURE 5.1 – Exemple de nœuds utilisés pour émettre et recevoir une interaction

de réception d'interaction est déclenché lorsque l'agent a été ciblé au moins une fois par l'interaction concernée par le nœud. En sortie du nœud *Receive Attack*, il y a deux valeurs : la première est un tableau contenant tous les agents ayant émis l'interaction concernée en ciblant l'agent courant, et ce, durant la frame courante. La seconde valeur est un tableau des forces que chaque agent transmet lors de l'émission de l'interaction. Il est alors possible de parcourir toutes les interactions d'un même type reçues durant une même frame.

5.4/ ANALYSES

Cette approche de gestion des interactions permet de diminuer le nombre de scans de l'environnement par les agents sans impacter le nombre d'interactions émises : Si les agents sont actifs constamment, ils effectueront le même nombre de scans de leur environnement que les agents dans une approche classique. Dès que des agents sont passifs, il y a une réduction du nombre de scans.

Nous avons des perspectives d'avenir pour cette approche. Nous allons chercher notamment à améliorer le parcours des tableaux d'enregistrements d'interactions que les agents émettent à chaque frame. Le fonctionnement en tableau n'est pas satisfaisant en termes d'accès. La première idée était de remplacer ces tableaux par des *map* permettant un accès en lecture plus rapide. Mais les axes les plus prometteurs pour ces tableaux seraient de les transformer en matrices. Différents travaux de recherche ont permis le développement de matrice d'interactions :

L'approche IODA [52] propose une matrice d'interactions permettant de paramétrer les interactions qu'une instance d'un type d'agent peut émettre vers une autre instance d'un type d'agent. Cette matrice n'a pas le même objectif que les tableaux de l'approche SAMP (qui gèrent les interactions à envoyer et non les interactions possibles entre agents) mais l'idée d'une matrice est prometteuse.

Nous avons eu des échanges lors des JFSMA 2018 durant lesquelles nous avons présenté l'approche décrite dans cette section. Une discussion nous a guidés vers l'approche *MIC** de Abdelkader Gouaïch, Fabien Michel et Yves Guiraud [41] qui propose l'utilisation de trois matrices d'interactions. Ces trois matrices permettent aux agents de se situer dans l'environnement et d'exposer les résultats de leurs perceptions. L'environnement de *MIC** est discret ce qui ne permet pas de l'appliquer facilement dans le domaine du jeu vidéo. De plus, il ressort de nos premières réflexions que de nombreuses informations et actions seraient dupliquées entre le moteur physique présent dans SAMP

et l'environnement de déploiement de *MIC** (déplacements des agents, positionnements, ...). Cependant, l'utilisation de matrices semble être une solution très intéressante pour améliorer la manière dont les interactions à émettre sont stockées par les agents.

Nous exposons plus tard dans ce mémoire, entre autres, les expérimentations et les validations de cette approche. Il est notamment intéressant de s'attarder sur l'efficacité de cette approche, mais aussi du coût en mémoire des tableaux d'interactions.

Le chapitre suivant décrit comment nous générons et exécutons les SMA modélisés dans SAMP.

6

GÉNÉRATION ET EXÉCUTION

Une fois que tous les agents, leurs interactions, leurs compétences et leurs comportements ont été modélisés et instanciés, il est possible d'exécuter le SMA. Pour l'exécution du SMA modélisé avec SAMP, nous avons étudié deux possibilités.

La première consistait en la création d'une machine virtuelle dans laquelle les modèles pouvaient être exécutés. Cette solution n'est pas la plus efficace pour notre objectif d'optimisation des SMA. Le fonctionnement d'une machine virtuelle consomme des ressources en plus du système qu'elle exécute. Cette solution a donc rapidement été abandonnée aux profits de la seconde.

La seconde solution qui s'est alors imposée à nous était la génération de code dans un langage natif, compilable et exécutable. Dans le domaine des jeux vidéo, et particulièrement quand il s'agit de console de jeux, les API fournies par les constructeurs de périphériques sont principalement développées en C++ et/ou compatibles C++. De plus, Shine Engine est lui aussi développé en C++. Ceci a guidé notre choix final : les SMA modélisés dans SAMP doivent être transformés en code en C++ qui sera compilé et exécuté.

Ce chapitre explique comment le code est généré à l'aide de SAMP et comment SAMP fait appel à des outils externes pour compiler et exécuter ce qui a été généré. Nous commençons par décrire comment les règles syntaxiques des modèles de SAMP sont vérifiées afin d'empêcher les utilisateurs de faire des *erreurs* de modélisation. Nous continuons en expliquant comment la génération du code est faite puis nous terminons en expliquant comment les utilisateurs peuvent exécuter les SMA générés afin de les tester ou de les intégrer à un jeu.

6.1/ LES TRANSFORMATIONS MODEL2MODEL ET MODEL2TEXT

Les comportements modélisés dans SAMP respectent le méta-modèle que nous avons défini (figure 4.10). Il est nécessaire de vérifier que les vues modélisées par les utilisateurs respectent les règles imposées par le méta-modèle. Pour nous assurer de ceci, nous avons décidé d'utiliser une approche *model to model* (M2M). Nous avons aussi cherché comment réaliser la génération du code à partir des éléments modélisés par les utilisateurs. Pour cela, nous avons utilisé une approche basée sur les transformations *model to text* (M2T).

Dans cette section, nous abordons ces concepts afin de savoir s'il peut se révéler utile

de les mettre en place dans SAMP.

6.1.1/ MODEL TO MODEL

Les transformations *model to model* (M2M) ont pour objectif de permettre de convertir un modèle en un autre modèle. Des travaux de recherche ont été menés afin de standardiser et améliorer le domaine des transformations de modèles. Le consortium Object Management Group (OMG), qui a pour but de standardiser un grand nombre de domaines de l'informatique et de l'industrie, propose MetaObject Facility (MOF) [44], un méta-modèle standard. Le standard *UML* est lui-même basé sur le standard *MOF*. Atlas Transformation Language *ATL* [48] s'inspire des travaux d'OMG sur les transformations.

Ces outils de transformations de modèles permettent d'automatiser les transformations de modèles. Nous ne cherchons pas à automatiser la transformation de notre méta-modèle SAMP-M en un autre modèle, mais à vérifier que les modèles produits par les utilisateurs respectent les règles définies par SAMP-M.

Il existe de nombreux travaux ayant pour objectif la vérification des transformations [13]. Certains travaux proposent une méthodologie pour effectuer une vérification manuelle des transformations [73]. D'autres travaux de recherche ont permis de développer des outils et algorithmes permettant l'automatisation de ces tests. Dans ces travaux, Erwan Brottier [22] définit un procédé pour automatiser les tests des données générées. Il explique qu'il y a 3 étapes dans ce procédé :

1. La décomposition du méta-modèle en plusieurs partitions représentant les types simples et les cardinalités du méta-modèle ;
2. A partir des partitions générées, la création de fragments de modèle qui peuvent être testés indépendamment des autres ;
3. La création de modèles valides pour les tests des fragments de modèles précédemment générés.

La spécificité de notre système est que les vérifications doivent se faire en temps réel. A chaque fois que l'utilisateur veut réaliser une action, nous le notifions si l'action souhaitée (ne) peut (pas) être réalisée directement. Une deuxième passe est effectuée lorsque l'action a été réalisée afin de vérifier si une erreur n'a pas été indirectement causée par cette action. C'est en partie pour ces raisons que nous avons décidé de développer notre propre système de vérification directement intégré à SAMP, ne requérant aucun outil externe. Cependant, nous nous sommes basés sur l'approche M2M pour réaliser notre propre développement.

6.1.2/ MODEL TO TEXT

La génération *Model to text* (M2T) permet de convertir un modèle en texte. La version de M2T qui nous intéresse ici est le *Model to Code* (M2C) permettant de convertir un modèle en code respectant la syntaxe d'un langage précis.

Tout comme les transformations M2M, il existe de nombreux outils et algorithmes transformant des modèles en code. On peut notamment citer MOFScript [63] qui définit des règles de génération de code basées sur les modèles *MOF*. La méthodologie de MOFScript est de rendre chaque élément du modèle indépendant des autres et de générer le

code correspondant à chaque élément indépendamment des autres. MOFScript définit des règles de transformation pour chaque élément. MOFScript possède un outil permettant l'édition de modèle MOFScript et la transformation en code.

La complexité liée aux transformations nécessaires de nos modèles pour pouvoir utiliser des outils externes, nous a décidé à développer notre propre mécanisme de transformation *M2C* directement dans SAMP. Cependant, nous avons adopté la méthodologie de MOFScript qui est d'appliquer les règles de transformation sur des éléments rendus indépendants les uns des autres où les seuls liens sont les transitions entre les éléments.

Dans la suite de ce chapitre, nous expliquons comment nous avons mis en place le contrôle des erreurs pour accompagner la modélisation faite par les utilisateurs, la génération du code à partir des vues, le paramétrage défini dans SAMP-E ainsi que la compilation de ce code en un plugin compatible avec Shine Engine.

6.2/ LE CONTRÔLE DES ERREURS

Dans la sous-section 4.6.2, nous avons vu les règles définissant les possibilités offertes aux utilisateurs pour modéliser des comportements. Ces règles permettent d'éviter que les comportements modélisés soient incohérents avec le méta-modèle de SAMP, et ainsi que le code puisse être généré.

Afin de guider au mieux les utilisateurs, SAMP propose une vérification en deux étapes afin de repérer les erreurs et notifier les utilisateurs de la présence de ces erreurs par le biais de son interface graphique. Il est à noter que SAMP permet d'ajouter dans une vue seulement les nœuds autorisés dans cette vue. Nous ne considérons pas cette fonctionnalité comme faisant partie du système de contrôle des erreurs, mais il permet d'éviter que des erreurs dans la syntaxe des comportements modélisés ne soient créées.

La première étape consiste en une analyse des erreurs à la volée. Lorsqu'un utilisateur cherche à réaliser une action qui se révèle être interdite, SAMP ne permet pas de valider l'action. Par exemple, il peut s'agir d'empêcher un utilisateur de créer un lien entre deux entrées ou deux sorties de nœuds, de créer un lien entre une entrée et une sortie dont les valeurs ne sont pas compatibles (et qu'il se révèle impossible de convertir le type de la valeur source dans le type de la valeur cible). Cette première étape permet de vérifier les actions qu'un utilisateur cherche à réaliser, mais elle ne vérifie pas les erreurs indirectement créées par ces actions.

C'est pourquoi nous avons mis en place une deuxième étape de vérification sur toute la vue une fois qu'une action a été validée. Cette étape réalise une vérification de tout le modèle courant. Les erreurs relevées par cette étape peuvent comprendre, par exemple, que la suppression d'un nœud fasse qu'une entrée d'un autre nœud se retrouve sans valeur.

L'interface graphique de SAMP permet de notifier l'utilisateur des erreurs présentes lorsqu'il est en cours de réalisation d'une action ou des erreurs présentent dans le modèle courant. Chaque notification est visible dans une fenêtre dédiée en lien avec un affichage particulier sur le modèle, à l'emplacement de l'erreur. Il est alors simple pour l'utilisateur de savoir où sont les erreurs et d'obtenir des informations complémentaires sur ces erreurs afin de pouvoir les corriger sans trop de difficulté.

Nous avons dressé la liste de toutes les erreurs qu'un utilisateur peut créer lorsqu'il mo-

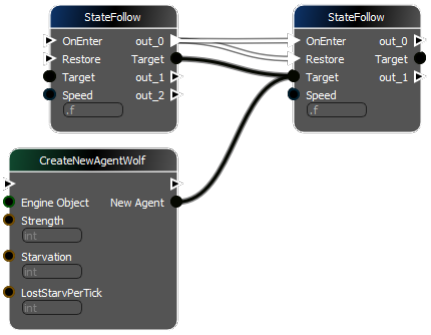

FIGURE 6.1 – Liens impossibles 1

délise un comportement dans SAMP. D'abord, les erreurs détectées lors de la première phase :

- Relier plusieurs sorties de valeurs sur une même entrée de *valeur*. Car dans ce cas, il serait impossible de déterminer quelle valeur récupérer (figure 6.1) ;
- Avoir plusieurs liaisons sur une même sortie d'exécution. Ceci afin de ne pas avoir plusieurs flux d'activation actifs en même temps (figure 6.1) ;
- Relier une sortie de *valeur* à une entrée d'*exécution* ou une sortie d'*exécution* à une entrée *valeur*. Ces deux types d'entrées-sorties ont des rôles totalement différents et ne peuvent pas être connectés entre eux (figure 6.2) ;
- Relier (directement ou indirectement) une sortie d'exécution d'événement externe d'un nœud *state* à une entrée d'exécution d'un nœud *state* sans passer par l'entrée d'exécution d'un nœud *event*. La règle citée précédemment est valable ici aussi. Il est nécessaire, pour passer d'un état à un autre, qu'un événement soit déclenché (figure 6.3(*i*)) ;
- Relier (directement ou indirectement) une sortie d'*exécution* de type événement interne d'un nœud *state* à un nœud *event*. La règle est qu'un agent ne réagit qu'à un seul événement en même temps et que les événements concurrents peuvent être priorisés (figure 6.3(*ii*)) ;
- Relier une sortie de *valeur* de type *T* à une entrée de *valeur* de type *V* et que *T* ne peut être converti [1] en *V*. Ceci afin d'éviter toute erreur de conversion de type lors de la compilation (figure 6.3(*iii*)) ;
- Relier deux entrées ou deux sorties entre elles afin d'éviter des boucles lors de la génération de code (figure 6.4) ;
- Supprimer un nœud qui ne peut pas l'être : Le nœud *exit point* d'une vue *événement* ou le nœud *OnEnter* d'une vue *comportement* sont obligatoires.

Les figures 6.1, 6.2, 6.3 et 6.4 exposent toutes les liaisons qu'il est interdit de réaliser lors de la modélisation d'un comportement dans SAMP-E et listées précédemment.

Ci-dessous, nous énonçons les anomalies détectées lors de la seconde phase :

- Vérifier que toutes les entrées de *valeur* possèdent une valeur. il est à noter que SAMP sait si une entrée possède une valeur par défaut, dans ce cas, si aucune valeur n'est indiquée, ce n'est pas une erreur qui est affichée, mais simplement un avertissement ;

1. SAMP permet de savoir si un type peut être converti, en un autre type, par héritage ou par une surcharge d'opérateur.

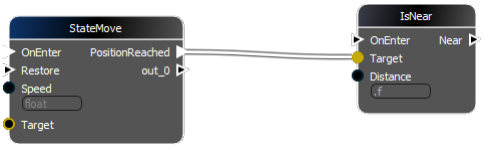

FIGURE 6.2 – Liens impossibles 2

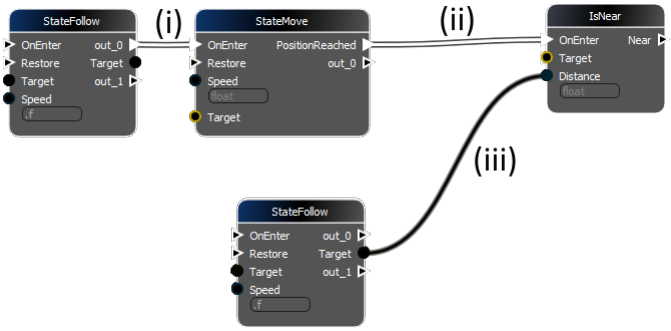

FIGURE 6.3 – Liens impossibles 3

— Vérifier que la valeur d'une sortie *valeur* est bien générée quand on la requiert. Par exemple, si une valeur provenant d'un nœud *fonction* est utilisée, mais que ce nœud *fonction* n'a pas encore été activé au moment où sa valeur est requise, une erreur est relevée.

6.3/ GÉNÉRATION DU CODE

Pour la génération de code, la figure 6.5 présente les quatre interfaces utilisées par l'ensemble des éléments générés par SAMP. Chaque élément composant les SMA modélisés dans SAMP hérite d'une de ces interfaces :

1. *CShAgentInstance*. Chaque agent crée dans SAMP génère une classe héritant de l'interface *CShAgentInstance* ;
2. *CShSkills*. Chaque compétence génère une classe héritant de l'interface *CShSkills* ;
3. *CShInteraction*. Chaque interaction génère une classe héritant de l'interface *CShInteraction* ;
4. *CShModel*. Chaque vue modélisée dans SAMP génère une classe héritant de l'interface *CShModel* ;

Chaque classe *CShAgentInstance* possède un tableau de *CShSkills* qui correspond aux compétences que l'agent maîtrise. Le fait de maîtriser des compétences permet à l'agent d'émettre ou recevoir des interactions qui sont stockées dans deux tableaux de *CShInteractions* (un pour les interactions qu'il peut émettre et l'autre pour les interactions qu'il peut recevoir). Chaque agent peut exécuter un comportement (représenté par la classe *CShBehavior*) et autant d'altérations qu'il veut (représenté par la classe *CShAlteration*).

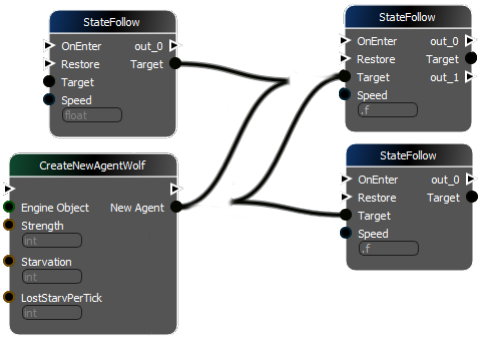

FIGURE 6.4 – Liens impossibles 4

6.3.1/ LES CLASSES *CShAgentInstance*, *CShSkills* ET *CShInteraction*

Les classes héritant de *CShAgentInstance*, *CShSkills* et *CShInteraction* sont générées à partir des informations issues des différentes fenêtres de paramétrages de SAMP. Nous allons aborder les points qui nous semblent les plus intéressants quant à ces générations de classes.

Les agents de SAMP possèdent des propriétés et celles-ci peuvent être accessibles aux autres agents. Pour rappel, nous avons mis en place un facilitateur qui permet à chaque agent de décider comment il partage ses propriétés aux autres agents (lesquelles ? et à qui ?).

Chaque classe agent est générée avec deux méthodes pour chaque type de propriétés qu'un agent possède. La première permet de récupérer une propriété d'un certain type (*getter*) et la seconde permet de changer la valeur des propriétés pour un type (*setter*). Ces méthodes possèdent toutes les mêmes types de retours et d'arguments :

```
ESetGetReturn GetProperty_XXX(CShIdentifier PropertyIdentifier,
                    CShAgentInstance * pAsker,
                    XXX & value);
```

```
ESetGetReturn SetProperty_XXX(CShIdentifier PropertyIdentifier,
                    CShAgentInstance * pAsker,
                    const XXX & value);
```

En argument de ces méthodes, on trouve :

— *PropertyIdentifier* qui est un *identifier* de la propriété [2] ;

— *pAsker* qui est de type *CShAgentInstance* et qui représente l'agent qui fait la demande d'accès ou de modification sur la propriété ;

— *value* qui est du type de la valeur à récupérer ou modifier. Dans la méthode pour accéder à la propriété (*getter*), cette valeur est paramétrée dans la méthode `GetProperty_XXX` seulement si le facilitateur de l'agent décide de transmettre cette valeur à l'agent requérant. Dans le cas contraire, elle n'est pas modifiée .

La valeur de retour de ces méthodes est une énumération qui peut prendre les différentes valeurs suivantes :

2. Chaque propriété générée possède un *identifier* sous forme d'entier permettant une recherche plus rapide qu'avec une chaîne de caractères.

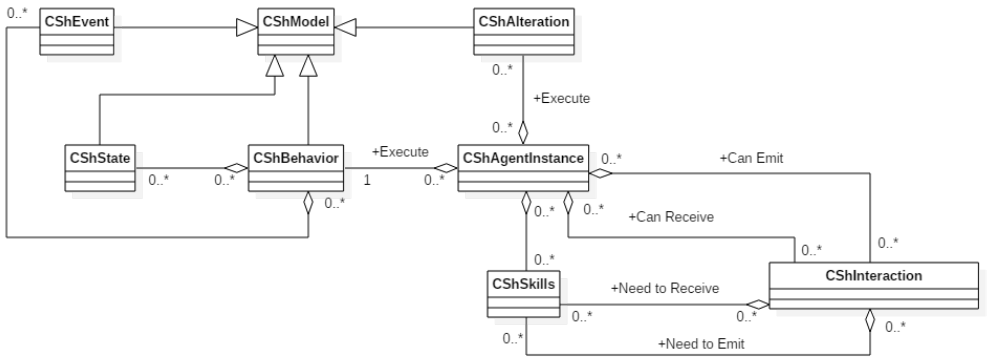

FIGURE 6.5 – Diagramme de classes de SAMP-X

— `OK` dans le cas où l'accès ou la modification ont été acceptés par le facilitateur de l'agent. Il est à noter que le facilitateur peut décider de renvoyer une valeur approximative de la propriété même s'il renvoie `OK` (Dans le cas d'une coopération truquée par exemple) ;

— `NOT_ALLOWED` dans le cas où l'accès ou la modification ont été refusés par le facilitateur de l'agent ;

— `NOT_EXIST` dans le cas où l'agent ne possède pas la propriété indiquée en argument. Le facilitateur peut, dans le cas où la propriété n'existe pas, décider de retourner la valeur NOT_ALLOWED plutôt que NOT_EXIST.

Tous ces paramétrages doivent, à l'heure actuelle être fait directement dans le code. Mais pour faciliter l'utilisation par des utilisateurs novices, nous allons développer une interface permettant de paramétrer les facilitateurs des agents. Dans notre exemple, un agent *mouton* possède ces deux méthodes pour accéder et modifier les propriétés de type entier (`int`) comme présenté dans le code suivant :

```
ESetGetReturn GetProperty_Int(CShIdentifier PropertyIdentifier,
                    CShAgentInstance * pAsker,
                    int & value)
{
  if (pAsker.type() == "sheep")
  {
    if (PropertyIdentifier == "starvation")
    {
      value = self.starvation;
      return(OK);
    }
    else if (PropertyIdentifier == "resistance")
    {
          value = self.resistance;
      return(OK);
    }
    else
    {
      return(NOT_EXIST);
    }
  }
  else
  {
    return(NOT_ALLOWED);
  }
```

```
}

ESetGetReturn SetProperty_Int(CShIdentifier PropertyIdentifier,
CShAgentInstance * pAsker,
int value)
{
  if (pAsker.type() == "sheep")
  {
    if (PropertyIdentifier == "starvation")
    {
      self.starvation = value;
      return(OK);
    }
    else if (PropertyIdentifier == "resistance")
    {
      self.resistance = value;
      return(OK);
    }
    else
    {
      return(NOT_EXIST);
    }
  }
  else
  {
    return(NOT_ALLOWED);
  }
}
```

Dans ces méthodes, on constate trois possibilités :

1. Si l'agent requérant (*pAsker*) est un agent de type Mouton, il peut accéder ou modifier les valeurs de l'agent possédant les valeurs :
   - Si la valeur existe, elle est modifiée ou assignée au paramètre *value* et la méthode retourne la valeur *OK* ;
   - Si la valeur n'existe pas, la méthode retourne *NOT_EXIST*.
2. Si l'agent n'est pas un agent de type mouton, la méthode retourne la valeur *NOT_ALLOWED*.

Les conditions de l'exemple sont basiques. Mais il est possible de faire des tests plus complexes en vérifiant, par exemple, si l'agent requérant possède une compétence particulière ou si un objet dans son inventaire est présent.

6.3.2/ LES CLASSES *CShModel*

Nous avons vu comment étaient générées les classes permettant le fonctionnement des compétences et des interactions des agents. Nous allons maintenant expliquer comment sont générées les classes permettant l'exécution des comportements.

Pour rappel, les comportements des agents sont modélisés dans des graphes de nœuds dans lesquels chaque nœud permet de définir une partie du comportement. Il existe 4 types de vues : les vues *comportement*, les vues *état*, les vues *événement* et les vues *altération*.

Les vues *événement* et *état* peuvent être utilisées dans d'autres vues. C'est pourquoi nous avons décidé de générer une classe C++ pour chaque vue. Lors des exécutions, il

suffira d'instancier la classe correspondant à la vue désirée afin de pouvoir exécuter le comportement qu'elle définit.

Chaque vue possède un ou plusieurs points d'entrée modélisés par des nœuds *entry point*. C'est à partir de chacun de ces nœuds que nous démarrons la génération de code de chaque vue. Nous parcourons le flux d'activation partant de chaque nœud *entry point* jusqu'à ce qu'un nœud *exit point* soit atteint ou que le flux d'activation soit stoppé.

Le fait d'exécuter la génération de cette manière permet de ne pas avoir à gérer les nœuds qui ne sont pas reliés à un flux d'activation. De plus, nous le verrons dans la suite de cette section, cela permet aussi la gestion de la portée des variables et la gestion de la fermeture des blocs.

Avant de faire ce parcours des flux d'activations, nous faisons une recherche des nœuds importants existants dans la vue. Il s'agit des nœuds *entry points* afin de connaître les points d'entrées de la génération ainsi que les nœuds *states* et *events* afin de pouvoir générer la déclaration, l'instanciation et l'initialisation des instances des classes de ces nœuds. Lorsqu'une instance est générée à partir d'un nœud *state* ou *event*, le nom de cette instance sera celui de l'*identifier* du nœud correspondant. En effet, dans SAMP-E, chaque nœud de chaque vue possède un identifier unique généré pour être compatible avec la syntaxe C++ de nommage des variables.

En plus des variables contenant les instances décrites précédemment, chaque classe héritant de *CShModel* possède une variable membre pointant sur l'instance de l'agent exécutant le comportement de la classe. Cette variable est appelée *AgentPerformer*.

Une fois la définition de la classe terminée, nous pouvons générer la déclaration de la classe ainsi que les définitions du constructeur, du destructeur et de la méthode d'initialisation de la classe. Une fois cela réalisé, nous pouvons passer générer les définitions des méthodes *OnEnter*, *OnLeave*, *OnPreUpdate* et *OnPostUpdate*.

Chaque nœud génère un code particulier pour chacune de ces méthodes. Cette spécificité permet une génération unitaire du code ce qui facilite le développement, la maintenance et la correction de la génération.

Dans la suite de cette section, nous décrivons le *Code Manager* permettant de gérer les transmissions des valeurs entre les nœuds au moment de la génération du code ainsi que la gestion de la portée des variables pour permettre la gestion des blocs de code.

6.3.3/ *Code Manager* : UN GESTIONNAIRE DE GÉNÉRATION DE CODE

Afin de nous aider dans le développement de notre générateur de code, nous avons mis au point un gestionnaire permettant de gérer divers aspects de la génération de code.

Lorsqu'un nœud possède une valeur de sortie de type retour (cf 4.6.2), il créé le code correspondant lors de sa génération. Pour ce faire, il demande au *Code Manager* de générer une variable du type de la valeur (avec un nom de variable inutilisé) et de lui assigner sa valeur de retour. Dans le même temps, il va ajouter des informations sur cette variable dans un tableau du *Code Manager* qui fonctionne un peu comme un dictionnaire de données. Ces informations sont :

— Le nom de la variable ;

— Le type de la variable ;

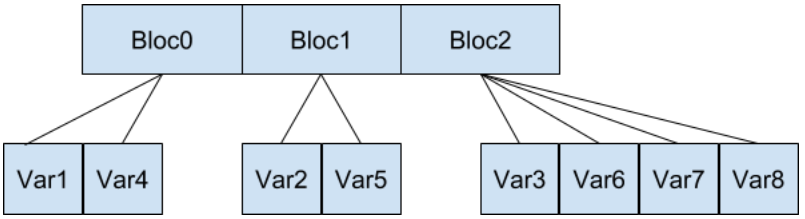

FIGURE 6.6 – Exemple du tableau de blocs et de variables du *Code Manager*

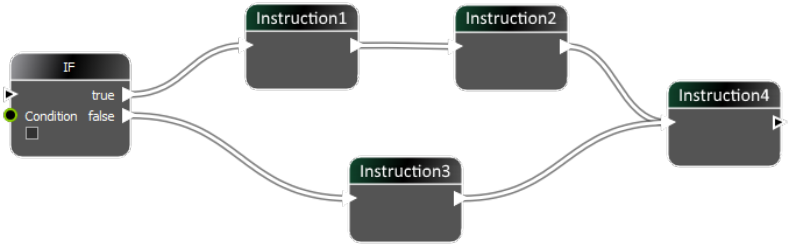

FIGURE 6.7 – Exemple de modélisation d'une condition *if-else*

— Le pointeur du nœud qui possède la valeur ;
— Le pointeur de la sortie de valeur du nœud.

Lorsqu'un nœud possède une entrée de *valeur*, reliée à une sortie de *valeur* d'un autre nœud, il va demander au *Code Manager* de lui retourner le nom de la variable correspondant. Si le *Code Manager* ne possède pas d'information sur cette variable, c'est que la variable requise n'a pas encore été générée. Cela peut être dû au fait que le nœud qui devait générer cette variable n'a pas encore été activé. Il s'agit d'une erreur de modélisation qui aura été détectée par le contrôle des erreurs (cf 6.2).

Les variables générées sont stockées dans un tableau du *Code Manager*. Il s'agit d'un tableau à deux niveaux (Figure 6.6) permettant de gérer la portée des variables dans les différents blocs d'instructions du code généré :

— Le premier niveau liste tous les blocs d'instructions créés. On considère qu'un bloc d'instructions à l'indice *i* dans le tableau est contenu dans le bloc d'instructions à l'indice *i-1*. Le bloc d'instructions d'indice 0 est le bloc d'instructions de plus haut niveau. La portée des variables dans ce bloc d'instructions est globale ;
— Le deuxième niveau permet de lister toutes les variables du bloc d'instructions.

Le dernier élément du premier niveau de ce tableau est toujours le bloc courant dans lequel le code est généré. Lorsqu'un bloc est ouvert par un nœud (*if*, *for*, ...), un nouvel élément est ajouté au premier niveau. Et lorsqu'une variable est générée, elle est ajoutée dans le dernier élément du premier niveau. Lorsqu'un bloc est fermé, le dernier élément de premier niveau est retiré du tableau.

En plus de permettre la gestion de la portée des variables, le *Code Manager* permet aussi de gérer quand doivent se terminer les blocs créés par des nœuds générant plusieurs flux d'activation. Un nœud générant plusieurs flux d'activation est, par exemple, un nœud *if* qui va générer un flux d'activation pour sa sortie *Alors* et un pour sa sortie *Sinon*. Le code manager est capable de détecter quand les deux flux d'activation se rejoignent et de fermer les blocs ouverts lors de la génération de chacun de ces flux d'activation.

Le but étant d'obtenir, depuis le modèle de la figure 6.7, un code de la forme :

```
if (condition)
{
    instruction1
    instruction2
}
else
{
    instruction3
}

instruction4
```

et de ne pas obtenir des codes de la forme :

```
if (condition)
{
    instruction1
    instruction2

    instruction4
}
else
{
    instruction3

    instruction4
}
```

6.3.4/ RÈGLES DE TRANSFORMATIONS

Dans cette sous-section, nous allons aborder les règles de transformations qui sont appliquées par les nœuds lorsque ceux-ci sont sollicités pour générer du code. Nous dressons la liste de chaque nœud et donnons, pour chacun, la ou les règles de transformation le concernant. Certains nœuds peuvent avoir plusieurs règles de transformation dépendant de la vue dans laquelle ils se trouvent ou dépendant de leurs paramètres.

Dans la suite de cette sous-section, nous écrivons les transformations dans un pseudo-code très proche du C/C++. La valeur de retour de chaque fonction de génération correspond au code qui sera généré. Nous commençons par décrire deux routines de génération de code commune à chaque nœud.

6.3.4.1/ UNE GÉNÉRATION COMMUNE

Chaque nœud possède sa propre fonction de génération de code. Cependant, il est possible de mutualiser certains traitements que nous avons placés sous formes de deux routines communes.

La première routine commune se trouve au début de la fonction de génération de code de chaque nœud possédant au moins une entrée activation. Cette routine permet de déterminer s'il faut fermer ou non le bloc courant, en coordination avec le *Code Manager* (cf 6.3.3), pour générer le code du nœud. Lorsque chaque nœud démarre la génération de son code, il interroge le *Code Manager* afin de savoir s'il fait partie du bloc courant ou non. S'il ne fait pas partie du bloc courant, le nœud va alors générer une fermeture de

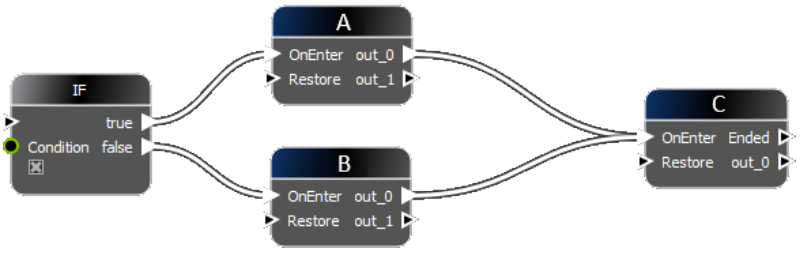

FIGURE 6.8 – Exemple de la génération de code avec un nœud *if*

bloc ({). Le *Code Manager* va alors lui indiquer s'il doit générer son code ou attendre. La figure 6.8 permet de mieux visualiser ce que fait ce code.

On remarque dans la figure 6.8 que le nœud *C* est connecté au nœud *if* deux fois. Lorsque la génération va se faire, on souhaite obtenir le code suivant :

```
if (...)  //Genere par le noeud if
{         //Ouverture de bloc par le noeud if
    A     //Genere par le noeud A
}         //Fermeture de bloc par le noeud C
else      //Genere par le noeud if
{         //Ouverture de bloc par le noeud if
    B     //Genere par le noeud B
}         //Fermeture de bloc par le noeud C
C         //Genere par le noeud C
```

On remarque dans ce code que même si le nœud *C* a été sollicité deux fois pour générer du code, il n'a généré qu'une seule fois son code et a fermé les blocs ouverts par le nœud *if*. En effet, lorsque le nœud *if* s'est généré, il active le nœud *A* pour sa génération qui ensuite active le nœud *C*. Le nœud *C* va demander au *Code Manager* ce qu'il doit faire. Le Code Manager va répondre au nœud *C* qu'il doit fermer le bloc et attendre avant de se générer. Le nœud *if* va alors activer le nœud *B* qui ensuite va activer à nouveau le nœud *C*. Le *Code Generator* va dire au nœud *C* de fermer le bloc et de générer son code.

Une seconde routine se trouve à la fin de chaque méthode de génération des nœuds possédants une et une seule sortie d'activation. Cette routine ne génère pas de code, mais elle permet de continuer le processus de génération. Cette routine se trouve dans une méthode appelée *CallNextNode* et qui retourne une chaîne de caractères contenant le code généré par le nœud suivant.

6.3.4.2/ LA MÉTHODE *ParameterGeneration*

Chaque nœud est associé à une classe qui possède une méthode utile à la génération de son code. Chaque type de nœud possède cette méthode, mais son contenu diffère en fonction du type du nœud. Cette méthode est appelée *ParameterGeneration* et son rôle est de rechercher dans le *Code Manager* (ou les générer et les ajouter dans le *Code Manager* s'ils n'existent pas encore) tous les paramètres nécessaires à la génération du code associé au nœud. Cette méthode possède 2 arguments de type chaîne de caractères et renvoie une chaîne de caractères.

```
string ParameterGeneration (string parameterDeclaration, string
    returnDeclaration)
```

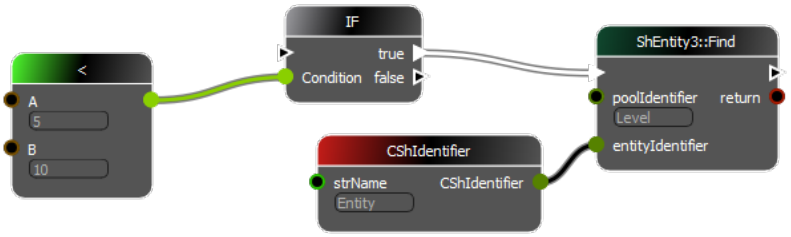

FIGURE 6.9 – Exemple pour l'algorithme de génération des paramètres des nœuds

— Le premier argument de cette méthode est *parameterDeclaration*. La méthode ajoute dans cet argument toutes les déclarations des variables qu'elle fait. Ces variables peuvent correspondre à des nœuds *value* connectés en entrée du nœud qui est en cours de génération de son code ou elles peuvent correspondre à des arguments de sorties (et non des valeurs de retour) du nœud.

— Le deuxième argument de cette méthode est *returnDeclaration*. La méthode ajoute dans cet argument la déclaration de la variable dans laquelle sera stockée la valeur de retour du nœud.

— La valeur de retour de cette méthode est une chaîne de caractères contenant les arguments d'entrée et de sortie du nœud. Cette chaîne est formatée pour être directement utilisée.

Chaque nœud possède cette méthode *ParameterGeneration* et son fonctionnement est le même pour tous les nœuds sauf les nœuds *comparator*. En effet, pour la majorité des nœuds, la génération des paramètres se fait en séparant les paramètres par des virgules. Pour ce qui est des nœuds *comparator*, la séparation n'est pas une virgule, mais un opérateur de comparaison booléenne séparant deux valeurs. La figure 6.9 expose un exemple possible de modélisation. Le nœud < est un nœud *comparator*, les nœuds `ShEntity3::Find` et *CShIdentifier* sont des nœuds *fonction*. `ShEntity3::Find` est une fonction statique et *CShIdentifier* est un constructeur. Une fois généré, on obtient le code (simplifié) suivant :

```
if (5 < 10)
{
  CShIdentifier var0("Entity");
  ShEntity3 return0;
  return0 = ShEntity3::Find(CShIdentifier("Level"), var0);
}
```

On constate qu'il y a deux générations de paramètres différentes en fonction du nœud généré. La variable *var0* est générée par le nœud *CShIdentifier* et la variable *return0* est générée par le nœud `ShEntity3::Find` pour stocker sa valeur de retour. On constate que dans l'appel à `ShEntity3::Find`, le premier argument est `CShIdentifier("Level")`. La méthode *ParameterGeneration* est capable de générer ce genre de code lorsqu'une valeur d'entrée d'un nœud est renseignée directement dans le nœud.

L'algorithme suivant exprime, en pseudo-code, comment est faite cette génération :

```
string ParameterGeneration (string parameterDeclaration, string
    returnDeclaration)
{
  string parameters;
```

```
  //
  // Generation parametres d'entree
  bool bFirst = true;
  foreach (self.aValueInput as value)
  {
    Variable variable;
    if (value.IsLinked())
    {
      Value linkedValue = value.GetLinkedValue();
      variable = CodeManager.FindVariableFromValue(linkedValue);

      if (variable == 0)
      {
        variable = CodeManager.CreateVariable(value);
        parameterDeclaration += variable.type + " " + variable.name + ";";
      }

      parameters += variable.name;
    }
    else
    {
      parameterDeclaration += value.Type + "(" + value.GetValueAsString() +")";
    }

    if (!bFirst)
    {
      if (self.NodeType() == e_type_comparator)
      {
        parameters += " " + self.comparatorType + " ";
      }
      else
      {
        parameters += ", ";
      }
    }

    bFirst = false;
  }

  //
  // Generation valeur de retour
  if (!self.aNodeOutput.IsEmpty())
  {
    Value value = self.aNodeOutput[0];
    Variable var = CodeManager.CreateVariable(value);
    returnDeclaration =  var.Type + " " + var.name;
  }

  return(parameters);
}
```

Dans cet algorithme, il est à noter le cas particulier suivant : `parameterDeclaration += value.Type + "(" + value.GetValueAsString() + ")";`. C'est cette instruction qui va générer le paramètre `CShIdentifier("Level")` de notre exemple. Il faut savoir que seules les entrées de valeurs dont le type est un type primaire ou un type *CShIdentifier* ou *CShString* (deux types de valeurs propres à Shine Engine) peuvent avoir une valeur renseignée dans une zone de texte. Par exemple, dans la figure 6.9, le nœud `ShEntity3::Find` possède une valeur renseignée dans une zone de texte, car il s'agit d'une valeur de type *CShIdentifier*.

Nous allons maintenant décrire les règles de transformation de chaque nœud de SAMP.

6.3.4.3/ LES NŒUDS *Entry*

Le premier nœud que nous analysons est le nœud *entry*. Ce nœud ne génère pas de code. Il permet d'indiquer pour chaque vue par où doit commencer la génération. C'est pour cette raison qu'il est obligatoire comme indiqué dans la section 6.2.

6.3.4.4/ LES NŒUDS *Exit*

Le nœud *exit* possède deux règles différentes dépendant de la vue dans laquelle le nœud se trouve. On utilise un nœud *exit* dans trois vues différentes avec deux générations de code différentes. Les deux pseudo-codes suivants permettent d'expliquer comment est généré le code d'un nœud *exit* dans les vues *comportement* (**1**) et dans les vues *état* et *événement* (**2**) :

— Dans une vue *Comportement* (**1**) :

```
string Generation (void)
{
  string generer = "AgentManager::GetInstance()->AddToRemove(
      AgentPerformer);";
  return(generer);
}
```

Lorsqu'un nœud *exit* est rencontré lors de la génération du code d'une vue *Comportement*, SAMP-E génère un code permettant de détruire l'agent exécutant le comportement en question. *AgentManager* est un singleton permettant la gestion de toutes les instances des agents. C'est *AgentManager* qui décide du meilleur moment pour détruire les agents qui doivent l'être. Pour rappel, la mort d'un agent dans le système est différente de la destruction de son instance dans le système.

— Dans une vue *État* ou une vue *Événement* (**2**) :

```
string Generation (void)
{
  Valeur v = self.aValeur[0];
  identifier id = v.GetAsIdentifier();
  string generer = "return(" + id.GetAsInterger() + ");";
  return(generer);
}
```

Lorsqu'un nœud *exit* est rencontré dans une vue *État* ou *Événement*, SAMP-E génère un code permettant de quitter l'exécution du comportement courant. Pour ce faire, on génère un *return*. Dans le cas où il y aurait plusieurs nœuds *exit* et afin de les différencier, le *return* est appelé avec pour valeur de retour une conversion de l'*identifier* du nœud *exit* en entier.

6.3.4.5/ LES NŒUDS *event*

Du fait que SAMP est basé sur un système d'états et d'événements, le code du comportement de la figure 6.10 prend cette forme :

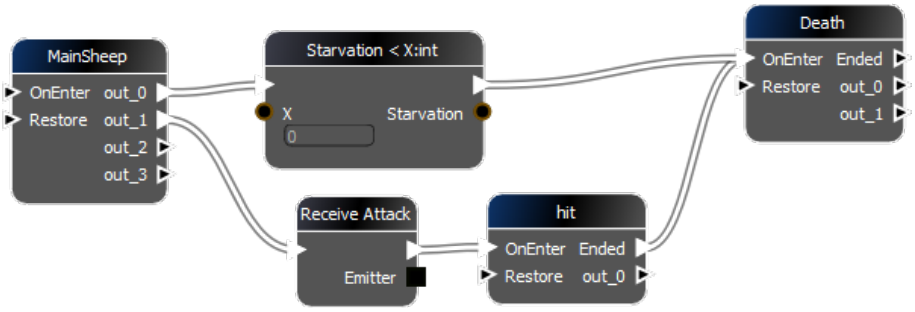

FIGURE 6.10 – Exemple d'un enchaînement d'état et d'événements.

```
if (self.MainShepp == self.currentState)
{
  if (self.Starvation_minus_x.IsTrue(0))
  {
    self.ChangeState(self.Death);
    self.Death.OnEnter();
  }
  else if (self.ReceiveAttack.IsTrue())
  {
    self.ChangeState(self.hit);
    self.hit.OnEnter();
  }
}
else if (self.hit == self.currentState)
{
  if (self.hit.InternalEvent() == "Ended")
  {
    self.ChangeState(self.Death);
    self.Death.OnEnter();
  }
}
```

La première partie de ce découpage est gérée par le nœud *state* lui-même. Il s'agit de la première condition permettant de savoir si l'état auquel est rattaché l'événement est l'état courant. Le nœud *event* quant à lui génère le code permettant de déterminer si sa condition d'activation est vérifiée. Ensuite, il active le nœud relié à sa sortie d'activation.

Le cas particulier des nœuds *state* avec un événement interne est géré par les nœuds *state* eux-mêmes (cf 6.3.4.6).

```
string Generation (void)
{
  string parameterDeclaration;
  string returnDeclaration;
  string parameter = ParameterGeneration(parameterDeclaration,
                                         returnDeclaration)

  string generer = parameterDeclaration;

  generer += "if(" + self.identifier + ".IsTrue(" + parameter + "));\n";
  generer += "{\n";
  generer += self.CallNextNode();
  generer += "}\n";
  return(generer);
}
```

6.3.4.6/ LES NŒUDS *state*

Lorsqu'un nœud *state* est rencontré dans une vue, SAMP-E génère un code permettant au comportement courant de changer d'état. Pour ce faire, on génère un appel à la méthode *ParameterGeneration*. Ensuite, on génère l'appel à *ChangeState*, qui est une méthode que possède chaque vue *Comportement* et *Etat* permettant de changer l'état courant de la vue. Puis nous générons l'appel à la méthode OnEnter du nouvel état courant, dans laquelle nous passons en paramètres les valeurs récupérées à l'aide de la méthode *ParameterGeneration*.

Ensuite, afin de générer le code sous la forme d'écrite dans la section 6.3.4.5, le nœud *state* génère une condition permettant de déterminer si l'état qu'il représente est l'état courant ou non.

Pour terminer, le nœud *state* active ses sorties d'activation avec deux cas possibles. Le premier, si la sortie est une sortie événement externe. Le second, si la sortie est une sortie événement interne (cf 4.6.5). Si la sortie activation est une sortie événement externe, le nœud connecté à cette sortie est activé. S'il s'agit d'une sortie événement interne, le nœud *state* génère une condition permettant de tester si l'événement est déclenché.

```
string Generation (void)
{
  string parameterDeclaration;
  string returnDeclaration;
  string parameter = ParameterGeneration(parameterDeclaration,
                                         returnDeclaration)

  string generer = parameterDeclaration;
  generer += "ChangeState(" + self.identifier + ");\n";
  generer += self.identifier + "->OnEnter(" + parameter + ");";

  generer += "if (self.currentState == self." + identifier + ")\n";
  generer += "{\n";

  foreach(self.aTriggerOutput as trigger)
  {
    if (trigger.isInternalEvent())
    {
      generer += "if (self." + self.identifier + ".InternalEvent()
                      == " + trigger.identifier + ")\n";
      generer += "{\n";
      generer += self.aTriggerOutput[0].GetLinkedNode().Generation() + "\n";
      generer += "}\n"
    }
    else
    {
      generer += self.aTriggerOutput[0].GetLinkedNode().Generation() + "\n";
    }
  }

  generer += "}\n"
  return(generer);
}
```

Les nœuds *state* ne peuvent pas avoir de valeur de retour, mais uniquement des arguments de sortie. C'est pourquoi la variable *returnDeclaration* n'est pas utilisée.

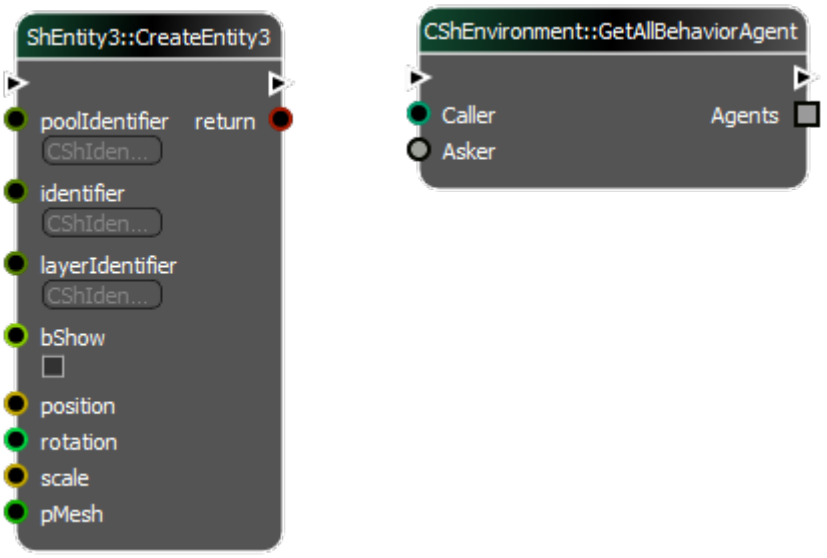

FIGURE 6.11 – Deux nœuds *fonctions* : à gauche statique, à droite non-statique

6.3.4.7/ LES NŒUDS *fonctions*

Lorsqu'un nœud *fonction* est rencontré dans une vue, SAMP-E génère un code permettant l'appel de la fonction correspondante. Pour commencer, on génère un appel à la méthode *ParameterGeneration*. Ensuite, s'il s'agit d'une méthode statique, on génère l'appel en concaténant le nom de la classe possédant la méthode avec le nom de la méthode puis avec les paramètres récupérés précédemment. S'il ne s'agit pas d'une méthode statique, on récupère le nom de l'objet qui appelle la méthode (le nœud connecté à l'entrée de valeur *Caller*[3] du nœud *fonction*). Puis on concatène ce nom avec le nom de la méthode et avec les paramètres récupérés précédemment.

```
string Generation (void)
{
  string parameterDeclaration;
  string returnDeclaration;
  string parameter = ParameterGeneration(parameterDeclaration,
                                         returnDeclaration)
  string generer = parameterDeclaration;

  if (!returnDeclaration.IsEmpty())
  {
    generer += returnDeclaration + "=";
  }

  string functionCall;

  if (self.IsStatic())
  {
    functionCall = self.GetClassName();
    functionCall += "::" + self.GetFunctionName();
  }
  else
  {
```

3. Les nœuds *fonctions* correspondant à des méthodes non statiques possèdent tous une entrée de valeur appelée *Caller* permettant d'indiquer au nœud quel objet va exécuter la méthode (cf Figure 6.11)

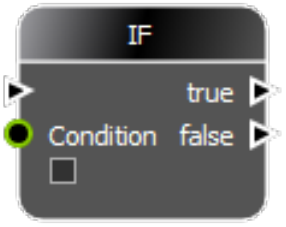

FIGURE 6.12 – Exemple d'un nœud *if*

```
    Variable var = codeManager.GetVariableFromInput("Caller");
    functionCall = var.GetName();
    functionCall += "." + self.GetFunctionName();
  }

  generer += "functionCall + (" + parameter + ");";

  generer +=  self.CallNextNode();

  return(generer);
}
```

6.3.4.8/ LES NŒUDS *if*

Lorsqu'un nœud *fonction* est rencontré dans une vue, SAMP-E génère un code permettant de générer un bloc *if*, et si besoin, un bloc *else*.

Pour commencer, la figure 6.12 expose un nœud *if*. Ce nœud contient une entrée d'*activation*, une entrée de *valeur* permettant de définir la condition ainsi que deux sorties d'*activation* : *true* qui est activée si la condition est vérifiée (correspond au bloc *Alors*) et *false* qui est activée si la condition n'est pas vérifiée (correspond au bloc *Sinon*).

Le principe de génération est simple, le nœud génère le code de la condition en fonction de ses paramètres d'entrée puis ouvre un bloc d'instructions. Une fois fait, il active le nœud relié à sa sortie d'activation *Alors*. Lorsque tout le flux d'activation de sa sortie *Alors* est terminé, et si besoin, il réalise les mêmes actions pour sa sortie *Sinon*. Il existe une troisième possibilité dans le cas où seule la sortie d'exécution *Sinon* est connectée à un autre nœud. Dans ce cas, le nœud *if* génère le bloc *if* en faisant la négation de la condition d'entrée.

```
string Generation (void)
{
  string parameterDeclaration;
  string returnDeclaration;
  string parameter = ParameterGeneration(parameterDeclaration,
                                         returnDeclaration)
  string generer = parameterDeclaration;

  // aTriggerOutput contient la liste des sorties d'activation.
  // Dans un noeud if, le premier element est la sortie Alors
  // et le second est la sortie Sinon
  if (self.aTriggerOutput[0].IsLinked())
  {
    generer += "if (" parameter + ")\n";
    generer += "{\n"
```

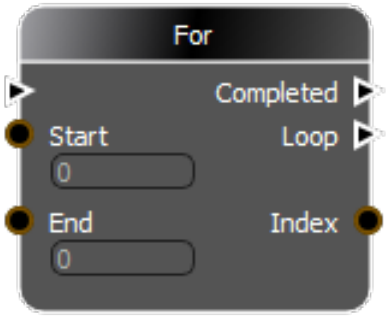

FIGURE 6.13 – Exemple d'une nœud *for*

```
    generer += self.aTriggerOutput[0].GetLinkedNode().Generation() + "\n";
    generer += "}\n"

    if (self.aTriggerOutput[1].Islinked())
    {
      generer += else + "\n";
      generer += "{\n"
      generer += self.aTriggerOutput[1].GetLinkedNode().Generation() +"\n";
      generer += "}\n"
    }
  }
  else
  {
    if (self.aTriggerOutput[1].Islinked())
    {
      generer += "if (! (" parameter + ") )\n";
      generer += "{\n"
      generer += self.aTriggerOutput[1].GetLinkedNode().Generation() +"\n";
      generer += "}\n"
    }
  }
  return(generer);
}
```

6.3.4.9/ LES NŒUDS *for*

Lorsqu'un nœud *for* est rencontré dans une vue, SAMP-E génère un code permettant de générer un bloc *for*. Pour commencer, la figure 6.13 expose un nœud *for*. On constate que ce nœud possède une entrée d'activation, deux entrées de valeurs, l'une pour indiquer sa valeur de départ (*start*) et l'autre pour indiquer sa valeur de fin (*End*). Ce nœud possède aussi une sortie de valeur permettant, durant chaque boucle du *for* de récupérer la valeur de l'index courant. L'incrément de ces boucles *for* est de 1 mais une version avec un entrée permettant de paramétrer l'incrément est en projet. Il possède également deux sorties d'activation : l'une qu'il faut relier aux nœuds modélisant le contenu du bloc *for* contenant les instructions à itérer (*Loop*) et l'autre qu'il faut relier aux nœuds modélisant le comportement exécuté après que le bloc *for* ait terminé son exécution (*Completed*)

Le principe de génération est le suivant : le nœud *for* récupère les paramètres pour générer la condition et crée une variable *Index* qui a pour portée l'intérieur du bloc *for*. Ensuite, le nœud génère une ligne contenant les conditions d'exécution du bloc puis ouvre le bloc. Il va ensuite activer le nœud relié à sa sortie d'activation *Loop*. Lorsque ce

flux d'activation se termine, le nœud va fermer le bloc et activer le nœud relié à sa sortie d'activation *Completed*.

```
string Generation (void)
{
  string parameterDeclaration;
  string returnDeclaration;
  string parameter = ParameterGeneration(parameterDeclaration,
                                         returnDeclaration)
  string generer = parameterDeclaration;

  Variable varStart = codeManager.GetVariableFromInput("Start");
  Variable varEnd = codeManager.GetVariableFromInput("End");
  codeManager.addVar("Index", "int");

  generer += "for (int Index = " + varStart.GetName() + ";
                 Index < " + varEnd.GetName() + "; ++Index)\n";
  generer += "{\n";

  if (self.aTriggerOutput[1].IsLinked())
  {
    generer += self.aTriggerOutput[1].GetLinkedNode().Generation();
  }

  generer += "}\n";
  codeManager.removeVar("Index");

  if (self.aTriggerOutput[0].IsLinked())
  {
    generer += self.aTriggerOutput[0].GetLinkedNode().Generation();
  }

  return(generer);
}
```

6.3.4.10/ LES NŒUDS *value*

Les nœuds *value* ne possèdent pas d'entrée ou de sortie d'activation mais ils possèdent une sortie de valeur. Le code généré par le nœud contient le nom de la variable. Les nœuds *value* ne sont activés que dans la méthode *ParameterGeneration*. Dans cette méthode, lorsqu'une entrée de *valeur* d'un nœud est reliée à une sortie de *valeur* d'un nœud *value*, la méthode *ParameterGeneration* va, si la variable concernée n'est pas accessible dans le bloc courant, demander au Code Manager de créer une variable, générer le code correspondant à cette création et stocker cette génération dans l'argument *parameterDeclaration*.

```
string Generation (void)
{
  string generer = self.identifier;
  return(generer);
}
```

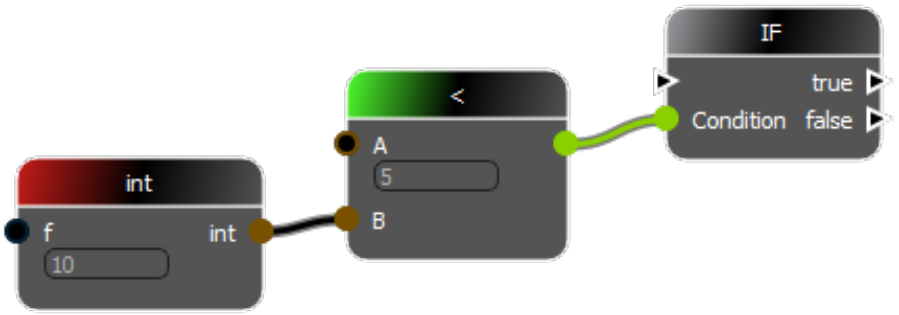

FIGURE 6.14 – Exemple d'un cas d'utilisation d'un nœud *comparator*

6.3.4.11/ LES NŒUDS *comparator*

Les nœuds *comparator* tout comme les nœuds *value* ne possèdent pas d'entrée ou de sortie d'activation. Lorsqu'ils sont activés, ils génèrent un code permettant de comparer les valeurs qu'ils possèdent en entrée. La figure 6.14 montre comment sont utilisés ces nœuds dans SAMP-E.

Sur cette figure on constate que le nœud comparator, lors de la génération de code du nœud *if*, va générer son code. Ce code sera en réalité généré par la méthode *ParameterGeneration* du nœud *if* qui va générer la condition requise par ce nœud en se servant des valeurs des nœuds *value* et *comparator*.

```
string Generation (void)
{
    string generer = self.comparatorType;
    return(generer);
}
```

Ce que l'on peut analyser de ces règles de transformations, c'est que la génération de code se fait de façon presque unitaire. Chaque élément de SAMP-E génère un petit bout du code du SMA modélisé. Lorsque les codes des agents, des interactions, des compétences et des comportements ont été générés, il ne reste plus qu'à générer le fichier CMake permettant la création d'un projet C++. Nous abordons dans la suite la création du projet C++ permettant la compilation du code généré.

6.4/ GÉNÉRATION DE PROJET ET COMPILATION

Une fois le code généré, il faut créer un projet C++ que nous pourrons compiler. Pour ses développements, l'entreprise Shine Research utilise l'*IDE* (Environnement de Développement intégré) Microsoft Visual Studio (*MVS*). Nous avons décidé d'utiliser le compilateur de *MVS* afin de compiler le projet créé avec CMake et généré par SAMP-E.

Lorsque la génération de code est terminée, SAMP-E génère une commande CMake afin de créer un projet C++ compilable. Il s'agit d'un projet Microsoft Visual Studio. Une fois le projet créé, SAMP-E génère une nouvelle commande pour, cette fois, compiler le code du projet Microsoft Visual Studio sous la forme d'une bibliothèque *C++* ;

Le projet est compilé pour être utilisé en tant que plugin dans Shine Engine. L'avantage de l'utilisation d'un plugin dans Shine Engine est qu'il est exécutable dans un jeu, mais

aussi dans l'éditeur Shine Editor.

Nous avons expliqué que la génération de code se faisait avec une approche unitaire. Chaque élément de SAMP-E génère juste ce qu'il représente. Cela facilite la maintenabilité du code en permettant de rapidement cibler l'origine d'éventuelles erreurs et de facilement ajouter de nouveaux éléments sans impacter les éléments déjà présents. Cependant, nous ne sommes jamais à l'abri d'une erreur causée par la modification du code de génération d'un élément de SAMP-E.

Afin de pallier au problème de régression et afin de valider le code généré, nous avons mis en place différents modèles de tests. Chacun de ces modèles permet de tester différentes fonctionnalités. Pour chaque modèle, nous indiquons s'il est conforme et quels résultats il doit donner. S'il n'est pas conforme, nous indiquons quelles erreurs doivent être levées. Le nombre de contraintes et d'erreurs possible est fini et pour chaque contrainte, le nombre de tests à réaliser pour vérifier qu'elles sont respectées est lui aussi fini. Nous avons donc un nombre de tests finis à réaliser pour vérifier que les contraintes sont respectées et nous couvrons chacun de ces tests. Cette couverture se rapporte aux contraintes et erreurs connues, que nous avons identifiées. Il est possible, par exemple, que des erreurs de compilation apparaissent dans du code généré pour des cas particuliers que nous n'avons pas encore rencontrés ni anticipés.

Au moment où nous écrivons ce mémoire, la validation doit se faire manuellement, car la comparaison de certains résultats n'a pas pu être automatisée. Nous sommes en cours de développement de tests unitaires sur *Code Testing Tool*, un outil de *Microsoft Visual Studio* permettant la création et l'exécution de tests unitaires au sein de l'environnement de développement.

6.5/ CONCLUSION

SAMP-E permet la modélisation de comportement à l'aide d'une interface graphique, mais permet aussi la génération sous la forme de code et la compilation en un plugin, compatible Shine Engine, des comportements modélisés. Une fois la modélisation réalisée, le contrôle d'erreur permet de relever toutes les erreurs présentes dans la modélisation. Sachant qu'un grand nombre d'erreurs ont déjà été empêchées lors de la modélisation par ce même contrôle d'erreur.

De par son format de génération unitaire, la génération de code est simple à maintenir et les tests de non-régression permettent de s'assurer de la validité du code généré par rapport aux modèles que l'on veut traduire. La compilation se fait en un clic et le contrôle d'erreur s'assure que lorsque la compilation est lancée aucune erreur de compilation ne peut apparaître. Tout au plus, des erreurs de liens peuvent être levées si les bibliothèques Shine Engine ne sont pas dans le bon dossier ou si elles ne sont pas à jour. Ces erreurs peuvent être dues à une mauvaise manipulation de l'utilisateur des fichiers du moteur Shine Engine.

III

EVALUATION DE LA MÉTHODE

7

CALCULS THÉORIQUES

Dans ce chapitre, nous allons évaluer, de manière théorique, l'efficacité de notre système d'interactions par rapport à l'approche classique. Nous commençons en exposant différentes formules permettant de calculer le nombre d'actions que réalisent les agents de chacune des approches pour réaliser les interactions. Nous continuons en définissant un exemple d'un jeu de stratégie basique auquel nous appliquons ensuite les formules afin d'obtenir des résultats exploitables.

7.1/ FORMULES

Pour valider notre approche, qui cherche à réduire le nombre des interactions entre agents, nous proposons d'évaluer de façon théorique les différentes approches. Dans cette section, nous décrivons les différentes entités manipulées pour cette comparaison :

- $\mathtt{A}$: l'ensemble de tous les agents.
- $\mathtt{A}_e$: l'ensemble des émetteurs, $\mathtt{A}_e \subseteq \mathtt{A}$
- $\mathtt{A}_{ea}$: l'ensemble des émetteurs actifs, $\mathtt{A}_{ea} \subseteq \mathtt{A}_e$
- $\mathtt{A}_t$: l'ensemble des récepteurs, $\mathtt{A}_t \subseteq \mathtt{A}$
- $\mathtt{A}_{ta}$: (resp. $\mathtt{A}_{tp}$) l'ensemble des récepteurs actifs (resp. passifs), $\mathtt{A}_{ta} \subseteq \mathtt{A}$, $\mathtt{A}_{tp} \subseteq \mathtt{A}$, $\mathtt{A}_{ta} \cap \mathtt{A}_{tp} = \emptyset$
- `Iem(ea)` : l'ensemble des interactions émises par un agent `ae`.
- `Ienv(ta)` : l'ensemble des interactions ayant pour récepteur l'agent `ta`.
- `Aenv(ea)` : l'ensemble des agents récepteurs passifs pour les interactions émises par l'agent `ea`.
- `OAs(i,ea)` : l'ensemble des agents récepteurs d'une interaction `i` de l'agent émetteur `ea` durant la frame précédente.
- `OI(a)` : l'ensemble des interactions sur lesquelles un agent `a` est déjà enregistré.
- `Si(i)` : l'ensemble des compétences nécessaires pour recevoir l'interaction `i`.
- `Ai(i)` : l'ensemble des agents récepteurs enregistrés sur l'interaction `i`.

Coût du test d'enregistrement vérifiant si un agent `ta` est enregistré sur l'interaction `i` :

$$CostRegTest(i, ta) = |Ai(i)| \tag{7.1}$$

Coût de l'enregistrement d'un agent a sur l'interaction i : `CostReg(i, a) = 1`

Coût des calculs des compétences requises détermine si un agent a possède les compétences requises pour réagir à *i*. Pour chaque compétence requise, nous considérons que la vérification si l'agent récepteur la possède est constante.

$$CostSkills(i,a) = |Si(i)| \tag{7.2}$$

Dans la suite, nous considérons le coût de recherche d'une interaction dans l'environnement comme identique, et ce, quelle que soit l'approche utilisée du fait de l'utilisation d'un moteur physique dans chacune des approches.

Nous commençons par établir pour les agents actifs, le calcul du coût de la recherche, de l'enregistrement et du désenregistrement des interactions. Posons :

— `ta` : un agent actif qui exécute la recherche d'interactions.

— i_n : une interaction de `Ienv(ta)`.

Le **coût du filtrage** détermine si `ta` peut réagir aux interactions. Ainsi, pour chaque interaction qui cible cet agent, pour laquelle il n'est pas encore enregistré, nous vérifions si cet agent a les compétences nécessaires pour réagir et si oui, il s'enregistre :

$$CostTAFilter(ta) = \sum_{i \in \texttt{Ienv}(ta) \setminus \texttt{OI}(ta)} (CostSkills(i,ta) + CostReg(i,ta)) + |Ienv(ta)| \tag{7.3}$$

Nous ajoutons à la formule $|Ienv(ta)|$ correspondant au coût du calcul de la différence ensembliste entre les deux ensembles des interactions.

Coût du désenregistrement concernant un **agent récepteur actif** `ta` et les interactions qui ne le ciblent plus (si `ta` est hors de portée de l'émetteur, si l'émetteur a arrêté d'émettre l'interaction, etc). `ta` se désenregistre des interactions présentes dans l'ensemble des interactions sur lesquelles il était enregistré lors de la frame précédente et dont il n'est plus cible dans la frame courante (comparaison entre les ensembles `OI(ta)` et `Ienv(ta)`). Nous considérons l'action de désenregistrement sur une table comme étant égale à 1. Nous ajoutons à la fin de la formule, le coût de la différence ensembliste :

$$CostUnregRec(ta) = \sum_{i \in \texttt{OI}(ta) \setminus \texttt{Ienv}(ta)} 1 + |OI(ta)| \tag{7.4}$$

Dans le pire des cas, où `ta` se désenregistre de l'ensemble des interactions, le coût sera 2 fois la cardinalité de OI(ta) : `CostUnreg(ta)` $\leq$ `2*|OI(ta)|`.

Pour un agent récepteur `ta`, le **coût total de ses interactions** est la somme du coût de filtrage des interactions donné par (7.3) et le coût de désenregistrement donné par (7.4) :

```
CostTA(ta) = CostTAFilter(ta) + CostUnregRec(ta)
```

Nous calculons maintenant le coût de fonctionnement des interactions pour un agent émetteur actif qui est composé de la recherche et de l'enregistrement sur les interactions et du désenregistrment des agents récepteurs de ces interactions.

Coût du désenregistrement depuis un **agent émetteur actif** `ea` sur un agent récepteur `ta` pour une interaction `i`. Ce coût inclut le test pour vérifier si l'agent est non ciblé par

l'interaction durant la frame courante (`i` $\in$ `OI(ta)` et `i` $\notin$ `Ienv(ta)`), et le coût de désenregistrement (si nécessaire) :

$$CostUnregEm(i, ea, ta) = 1 + 1 + 1 = 3 \tag{7.5}$$

Coût total du traitement des interactions pour un agent émetteur actif. Il est composé du coût de l'enregistrement de la nouvelle interaction (7.3) et du coût de désenregistrement (7.5) pour l'agent qui n'est plus ciblé par l'interaction durant la frame courante. A chaque fois, nous ajoutons le coût de la différence ensembliste :

$$\begin{aligned} CostEmitterAgent(ea) = \sum_{i_n \in \texttt{Iem}(ea)} (& \\ & \sum_{ta_r \in \texttt{Aenv}(ea) \setminus \texttt{OAs}(i_n, ea)} (CostSkills(i_n, ta_r) + CostReg(i_n, ta_r)) + |Aenv(ea)| \\ & + \sum_{ta_u \in \texttt{OAs}(i_n, ea) \setminus \texttt{Aenv}(ea)} CostUnregEm(i_n, ea, ta_u) + |OAs(i_n, ea)| \\ &) \end{aligned} \tag{7.6}$$

Nous définissons le **coût d'émission d'une interaction** $\texttt{i}_n$ de `Iem(e)` pour un agent émetteur `e` et un agent récepteur `ta` :

$$CostEmitInter(i_n, e, ta) = 1 \tag{7.7}$$

Le **coût total d'émission des interactions** est pour un agent émetteur `e` :

$$TotalCostCom(e) = \sum_{i_n \in \texttt{Iem}(e)} \sum_{ta_n \in \texttt{Ai}(i_n)} CostEmitInter(i_n, e, ta_n) \tag{7.8}$$

Nous pouvons donc définir le **coût complet** de la gestion **des interactions de notre approche** comme étant le coût de fonctionnement des interactions pour les agents récepteurs actifs ainsi que pour les agents émetteurs actifs et passifs. Pour les agents émetteurs, le calcul se fait en trois parties : une première qui va calculer le coût d'émission des interactions pour tous les agents récepteurs, une seconde qui calcule le coût de traitement des interactions pour les agents émetteurs actifs (7.6) et la dernière qui calcule le coût des émissions des interactions de l'ensemble des agents (7.8).

$$\begin{aligned} TotalCostInt = & \sum_{ta \in A_{ta}} CostTA(ta) \\ & + \sum_{ea \in A_{ea}} CostEmitterAgent(ea) \\ & + \sum_{e \in A_e} TotalCostCom(e) \end{aligned} \tag{7.9}$$

Maintenant, nous établissons le calcul du **coût complet** de la gestion **des interactions pour le système classique** dans l'environnement de simulation `env`, Il considère pour chaque agent du système, le coût de la vérification pour établir les interactions auxquelles il réagit (7.2) et le coût d'émission des interactions qu'il émet (7.7) :

$$TotalCostClassicSystem = \sum_{a_n \in A} (\sum_{i_n \in \texttt{Ienv}(a_n)} CostSkills(a_n, i_n) + \sum_{i_t \in \texttt{Iem}(a_n)} CostEmitInter(i_t, a_n, env)) \quad (7.10)$$

7.2/ RÉSULTATS

Dans cette section, nous présentons une instance des calculs définis précédemment. Pour illustrer notre démarche, nous commençons par décrire un exemple de SMA. Ensuite, nous analysons les résultats des calculs théoriques appliqués à ce système afin de comparer notre approche à l'approche classique. Nous étudions différentes valeurs de paramètres du SMA afin d'observer les tendances des résultats à travers les formules théoriques.

7.2.1/ JEUX DE STRATÉGIE EN TEMPS RÉEL

La première étape est de définir un système avec des valeurs réalistes, proches d'un jeu vidéo. Nous avons choisi de simuler un jeu de stratégie en temps (STR). Ce type de jeu vidéo met en scène différentes équipes qui s'affrontent entre elles à l'aide d'unités mobiles, de bâtiments de défense et de bâtiments de production.

Pour nos calculs, nous avons décidé d'opposer deux équipes. Chacune d'elle peut créer deux types de bâtiment et un type d'unité. Un agent *hutte* qui ne peut exécuter aucune action et est constamment passif (bâtiment de production). Un agent *tour* qui ne peut pas bouger, mais peut attaquer les ennemis entrant dans son champ de vision. Une tour est constamment passive même lorsqu'elle attaque. En effet, en attaquant, elle ne changera pas la perception que les autres agents ont d'elle ni sa perception des autres agents. Ce sont les agents actifs se déplaçant à proximité de la tour qui feront un scan de l'environnement pour détecter la tour et s'enregistrer dans sa table d'enregistrement ou les agents actifs sortant de son champ de vision qui se désenregistreront de sa table d'enregistrement.

Un agent *soldat* qui peut se déplacer pour se rapprocher des ennemis et les attaquer. Un soldat est passif quand il ne bouge pas et devient actif lorsqu'il se déplace. Pour optimiser ce système dans notre approche, nous considérons un soldat qui attaque comme étant passif (il ne se déplace pas durant son attaque).

Tous les agents peuvent être *vus* par les autres agents possédant la compétence *voir*. Tous les agents peuvent être *endommagés* par les agents possédant la compétence *attaquer*. Les propriétés des agents sont présentées dans la table 7.1.

Pour comparer les deux approches, nous analysons différentes configurations du système. Nous avons étudié l'impact de cinq paramètres du système :

— Le nombre d'agents de chaque type présent dans le système

— Le nombre d'agents actifs.

TABLE 7.1 – Propriétés des agents du STR

Agent	Etat	Compétence	Interaction
Hutte	Passif	Endommageable	Etre vu
Tour	Passif	Voir Endommageable	Etre vu Attaquer
Soldat	Actif Passif	Voir, Marcher Endommageable	Etre vu Attaquer

— Le pourcentage d'agents qui sont dans le champ d'action des interactions des autres agents.
— Le pourcentage d'agents déjà enregistrés sur une interaction à chaque frame
— Le pourcentage d'agents qui se désenregistrent d'une interaction à chaque frame.

Dans notre expérimentation, nous utilisons les valeurs suivantes pour paramétrer nos calculs : 1000 huttes, 2000 tours et 3300 soldats.

1. Nous fixons le nombre d'interactions actives pour un agent à la moyenne du nombre d'interactions. Dans notre exemple, le nombre d'interactions pour un agent est 1.84 = (1000*1+2000*2+3300*2)/6300.
2. Nous fixons le nombre de compétences pour un agent à la moyenne du nombre de compétences pour chaque agent. Dans notre exemple, le nombre de compétences par agent est 2.88 =(1000*1+2000*2+3300*4)/6300.
3. Nous décidons que chaque agent dans le champ d'action d'une interaction s'enregistre dessus.

7.2.2/ CALCULS ET ANALYSES

Nous définissons trois termes pour nos analyses :

Définition (Densité de population)**.** *La densité de population est le nombre d'agents qui peuvent interagir avec les autres à un instant t.*

Définition (Diversité de population)**.** *La diversité de population est le nombre d'agents de chaque type à un instant t.*

Durant nos expérimentations, quand nous changeons la diversité de population, le nombre d'agents de la simulation n'est pas impacté. Par exemple, si nous ajoutons 200 soldats, nous retirons 200 autres agents.

Nous paramétrons le taux de soldats actifs à 50% (1650 agents actifs et 4650 agents passifs)

Définition (taux d'enregistrement)**.** *Le taux d'enregistrement correspond au nombre de nouveaux agents qui cherchent à s'enregistrer eux-mêmes sur une interaction sur laquelle ils ne sont pas déjà enregistrés.*

Par exemple, si nous avons 100 agents et que le taux d'enregistrement est de 2%, cela signifie que 2 nouveaux agents seront enregistrés sur chaque interaction à chaque frame. Pour rappel, dans notre approche, un agent ne s'enregistre qu'une seule fois sur une interaction. Les agents vérifient s'ils possèdent les compétences nécessaires pour recevoir l'interaction. Changer le taux d'enregistrement permet de voir l'impact de ce filtrage.

7.2.2.1/ MODIFICATION DE LA DENSITÉ DE POPULATION

Nous commençons par étudier l'impact de la variation de la densité de population sur les deux approches. Nous faisons évoluer la densité de population de 1 à 125 agents. Nous calculons les résultats pour différents taux d'enregistrement.

Dans la figure 7.1 (ne montrant qu'une partie des résultats : densité de population supérieure à 75) nous pouvons voir que notre approche est meilleure que l'approche classique pour un taux d'enregistrement inférieur à 80%. Dans les faits, il est impossible d'avoir un taux d'enregistrement de 100% durant deux frames consécutives, car si tous les agents sont enregistrés sur une interaction durant la première frame, ils n'auront pas besoin de s'enregistrer à nouveau durant la seconde frame. Le plus mauvais taux d'enregistrement moyen possible est de 50% à chaque frame.

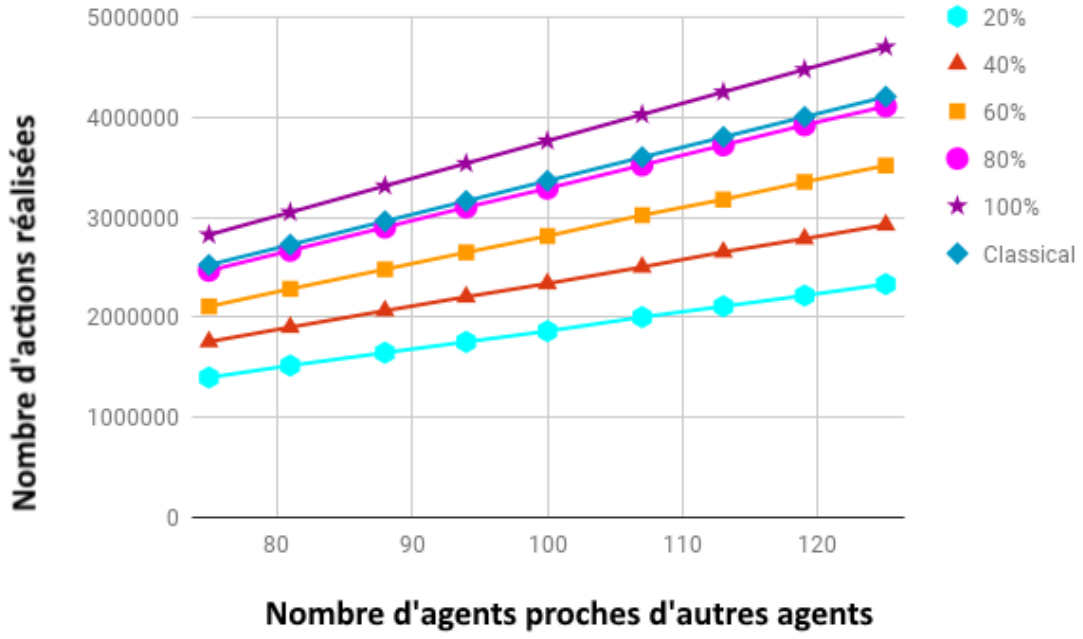

FIGURE 7.1 – Changement de densité de population.

Pour les calculs suivants, nous décidons d'utiliser le pire des cas de notre approche : un taux d'enregistrement de 100%. Ce paramétrage permettra d'évaluer l'impact des autres paramètres sur l'efficacité de notre approche comparée à l'approche classique, dans le pire des cas. Nous paramétrons la densité de population à 1% (62 agents), la valeur médiane de notre approche (1 à 125) .

7.2.2.2/ IMPACT DU NOMBRE D'AGENTS ACTIFS

Nous étudions l'impact du ratio d'agents actifs/passifs ainsi que l'impact d'un changement de configuration sur l'efficacité de notre système. Nous créons deux simulations : la première utilisant les paramètres précédemment définis et pour la seconde, nous donnons une compétence supplémentaire aux agents hutte et tour. Pour les deux simulations, nous faisons évoluer le ratio d'agent actifs/passifs de 0.03 à 1.10. Dans la fi-

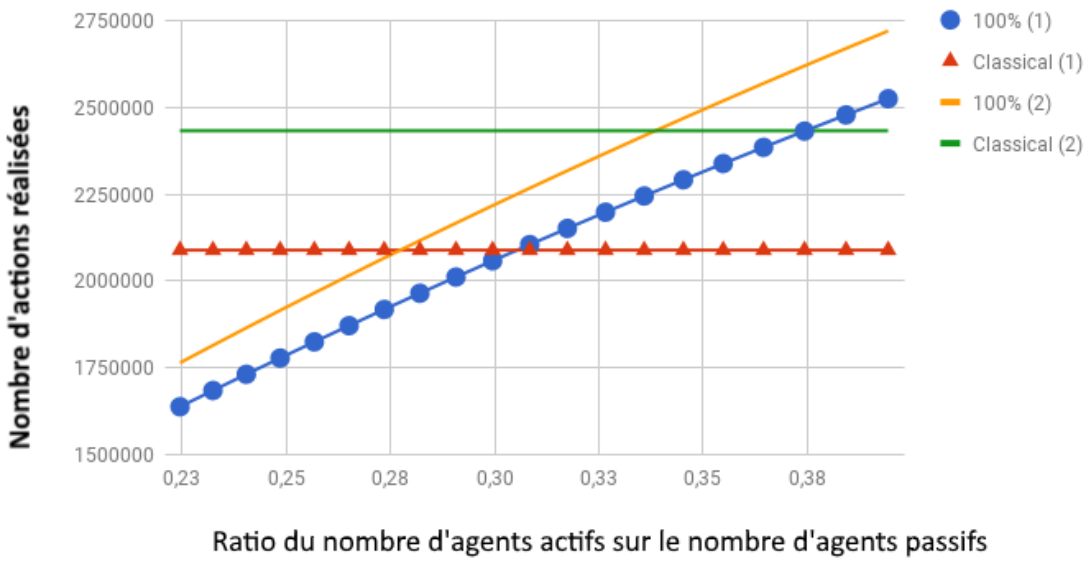

FIGURE 7.2 – Changement du ratio d'agents actifs/passifs

gure 7.2, les courbes montrent les résultats de la première simulation (1) et les résultats de la simulation avec une configuration modifiée (2). Pour clarifier les informations, nous avons décidé de ne montrer les résultats que pour un ratio d'agents actifs/passifs de 0.23 à 0.39 (offrant une meilleure visibilité aux croisements des courbes).

Nous pouvons d'abord constater que le ratio d'agents actifs/passifs a un impact sur l'efficacité de notre approche. Notre système est plus efficace quand ce ratio est inférieur à 0.31 pour (1) et 0.34 pour (2). Ces résultats sont cohérents pour un jeu vidéo dans lequel le nombre d'agents passifs (portes, coffres, arbres, ...) est souvent plus élevé que le nombre d'agents actifs.

Concernant le changement de configuration, nous pouvons constater qu'il est plus efficace que le système classique quand les agents possèdent plus de compétences. Ceci est dû au fait que, dans notre approche, la phase de filtrage n'est exécutée qu'une fois durant la durée d'une interaction et non à chaque frame comme dans l'approche classique.

7.2.2.3/ CALCULS POUR UNE SECONDE

Nous avons précédemment parlé de l'importance du respect du nombre de frames par seconde dans les jeux vidéo. Nous réalisons maintenant un calcul pour une seconde (30 frames). Rappelons qu'un humain concentré sur un stimulus réagira à ce stimulus [1] en 250ms (8 frames) et qu'un conducteur réagit à un danger en 1 à 2 secondes [58]. Un calcul sur une seconde semble être proche de la réalité.

Nous considérons qu'un agent s'enregistre lui-même ou il est enregistré par d'autres agents sur une seule interaction par seconde. Dans le système classique, tous les agents scannent l'environnement pour vérifier si des interactions pourraient les concerner. Dans notre approche, ces scans ne sont exécutés qu'au moment où l'émetteur débute l'émis-

1. https ://www.humanbenchmark.com/tests/reactiontime

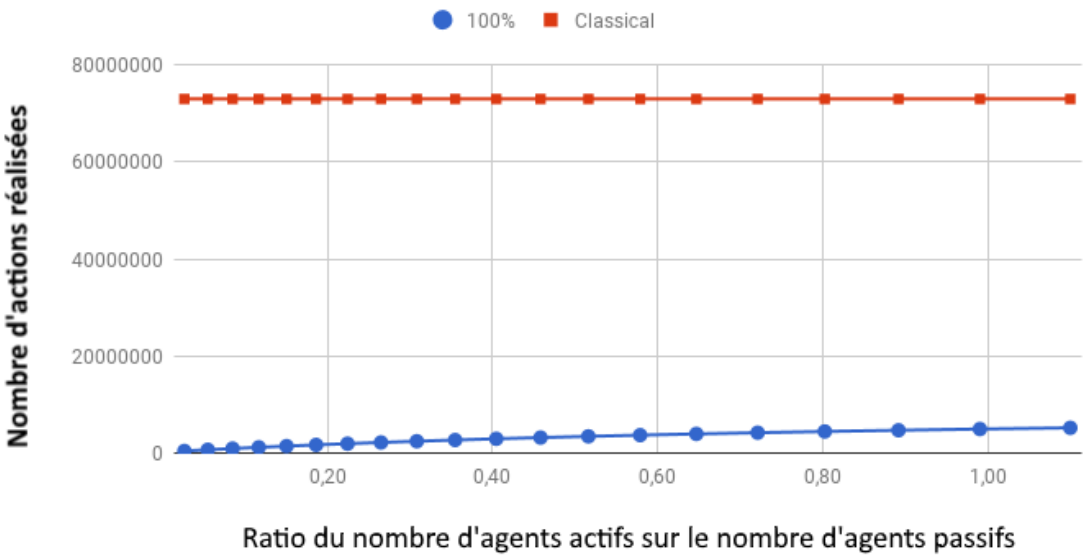

FIGURE 7.3 – Changement du nombre de soldats actifs avec un calcul sur 1 seconde.

sion de l'interaction ou quand un agent est actif. Dans les deux approches, les envois des interactions sont effectués à chaque frame. Nous faisons évoluer le nombre de soldats qui sont actifs et nous gardons la densité et la diversité de population fixe. Pour notre approche, nous exécutons le calcul pour un taux d'enregistrement de 100%. La figure 7.3 montre que, même dans le pire des cas, en temps réel, notre approche est plus efficace que l'approche classique.

7.3/ CONCLUSIONS

Les résultats théoriques obtenus à l'aide de ces calculs sont très encourageants. Ils permettent de constater que dans un système en temps réel, l'approche SAMP pour les interactions est plus efficace qu'une approche classique. Fort de ces résultats, nous avons décidé de réaliser des expérimentations pratiques afin de tester SAMP en conditions réelles et de comparer son efficacité par rapport à une approche classique.

8

RÉSULTATS PRATIQUES

Dans ce chapitre, nous exposons nos expérimentations. SAMP est développé afin d'être utilisé dans le domaine du jeu vidéo, mais son utilisation n'est pas restreinte à ce seul domaine.

Nous commençons en exposant les résultats d'expérimentations pratiques autour du système proies-prédateurs modélisé avec SAMP. Pour cela, nous présentons différentes méthodes d'expérimentation et expliquons comment nous avons mené ces expérimentations. Nous continuons en décrivant les conditions dans lesquelles les expérimentations ont été effectuées. Ensuite, pour chacune d'entre elles, nous présentons ses paramètres puis nous exposons et analysons ses résultats.

Nous exposons ensuite trois systèmes modélisés à l'aide de SAMP. Le premier est un jeu en 2D, le second est un SMA utilisé dans un projet nommé Silva Numerica qui est un *serious game*. Le troisième est un applicatif de communication de type lecteur de flux Rss utilisant SAMP pour gérer le comportement des éléments qui le composent. Le développement de ces trois systèmes a pour but d'expérimenter l'expressivité de SAMP, sa capacité à permettre la modélisation de SMA utilisables dans d'autres domaines que celui des jeux vidéo et la possibilité d'importer/exporter des éléments entre différents projets.

Nous concluons ce chapitre en analysant les résultats que nous avons obtenus.

8.1/ EXPÉRIMENTER

Afin d'organiser nos expérimentations, nous avons cherché ce qu'il existait dans l'état de l'art au sujet de l'organisation des expérimentations. Le livre de Pierre Dagnelie *Principe d'Expérimentation, Planification des expériences et analyse de leurs résultats* [25] nous a amené beaucoup de réponses.

Pierre Dagnelie décrit dans son livre qu'il existe différentes étapes à la réalisation d'expérimentations :

1. La définition du ou des buts ;
2. La définition du ou des facteurs dont on désire étudier l'influence ;
3. La définition des individus ;
4. La définition des observations ;

5. Les différentes affections que l'on donnera aux facteurs pour faire évoluer les expérimentations ;
6. Les analyses.

8.1.1/ NOS BUTS

Dans notre approche, nous cherchons à diminuer le nombre de scans de l'environnement que chaque agent réalise afin d'alléger le fonctionnement des interactions. Le but de nos expérimentations sera de comparer notre système à une approche classique. Nous chercherons aussi à étudier l'impact que peuvent avoir différents paramètres d'un SMA sur notre approche.

8.1.2/ QU'EST-CE QU'UN FACTEUR ? QUE SONT LES NIVEAUX ?

Dans son livre, Pierre Dagnelie explique que les facteurs sont *des séries d'éléments de même nature pouvant être comparés*. Il explique aussi que les éléments composants les séries sont appelés *variantes* ou *niveaux*. Le terme *variante* est plus adapté dans le cas de facteurs qualificatifs (dont les éléments ne peuvent pas être classés) et le terme *niveau* est plus adapté dans le cas de facteurs quantitatifs (dont les éléments peuvent être classés).

Dans nos expérimentations, nous avons isolé trois facteurs quantitatifs (le nombre d'agents, le nombre de scans effectués et le nombre d'interactions émises), nous utiliserons donc le terme *niveau*.

Les facteurs peuvent être contrôlés, non contrôlés ou constants. Un facteur contrôlé est un facteur qui est étudié au cours de l'expérimentation. Il est possible de fixer les valeurs de certains facteurs pour en faire des facteurs constants afin de ne pas multiplier le nombre d'éléments à étudier au cours des expérimentations. Nous verrons dans la suite de ce chapitre que nous aurions pu fixer le nombre d'agents constant, mais que cela se révélait inutile.

Ainsi, durant nos expérimentations, nous avons besoin d'étudier 3 facteurs : le nombre d'agents, le nombre d'interactions émises par agent et le nombre de scans de l'environnement effectués par chaque agent.

8.2/ CONDITIONS D'EXPÉRIMENTATION DU FONCTIONNEMENT DES INTERACTIONS

A l'aide de SAMP, nous avons généré quatre SMA exécutant chacun un système proies-prédateurs. Sur ces quatre systèmes, deux utilisent l'approche d'interactions SAMP et deux utilisent une approche classique. Pour chacune des approches, nous avons un système proies-prédateurs complet comme décrit dans notre exemple fil rouge et un système proies-prédateurs dans lequel les moutons ont été rendus aveugles afin d'identifier l'impact de la réduction du nombre de compétences sur le nombre de scans et d'interactions.

En effet, dans les SMA proies-prédateurs où les moutons sont aveugles, lorsqu'un mouton a faim, il patiente sur place 3 secondes avant de manger, quelle que soit sa position.

Il possède aussi un certain pourcentage de chance de se cloner sur place après avoir mangé (le mouton ne part plus en recherche d'un autre mouton pour se reproduire). Ici, le but n'était pas d'avoir un système réaliste, mais de tester l'impact du nombre d'interactions sur l'efficacité de notre approche par rapport à l'approche classique, mais aussi de tester cet impact entre deux SMA utilisant notre approche.

Du fait que les moutons ne maîtrisent plus la compétence *voir*, ils ne sont plus capables de recevoir les interactions émises par les agents herbes et les autres agents moutons.

Pour chacun des SMA générés, nous avons réalisé deux scènes permettant de représenter l'état de départ des SMA pour les expérimentations. La première avec un nombre réduit d'agents au départ de la simulation (89 agents [1]). L'autre avec un nombre beaucoup plus grand (1424 agents[1]). Ceci afin de tester l'impact du nombre d'agents sur les différents SMA générés. Dans chacune des expérimentations, la population évolue. Des agents se reproduisent, d'autres meurent. C'est la seule variable sur laquelle nous avons le contrôle, qui évolue et qui représente un intérêt pour nos résultats. Nous n'avons pas un contrôle direct sur les autres variables qui évoluent et qui représentent un intérêt pour nos résultats. C'est notamment le cas pour le nombre de scans réalisés par frame et le nombre d'interactions émises par frame.

Nous analysons, pour chaque expérimentation, la moyenne du nombre de scans effectués par chaque agent en fonction du nombre d'agents dans la simulation. Pour chaque expérimentation, nous affichons dans un même graphe les courbes correspondant aux valeurs analysées pour notre approche et l'approche classique. Chaque exécution dure 5000 frames [2].

Avant de commencer l'analyse des résultats de chaque expérimentation, il est bon de noter qu'entre les systèmes générés avec une approche SAMP et ceux générés avec une approche classique, très peu d'éléments changent. Dans l'approche SAMP, seuls les agents **actifs** scannent leur environnement à la recherche d'interactions qui les intéressent et à la recherche d'agents intéressés par leurs interactions. Ensuite, chaque agent émet des interactions à destination des agents présents dans leur table d'émission.

Dans l'approche classique, chaque agent scanne l'environnement à la recherche d'interactions pouvant l'intéresser. Une fois cela fait, ils n'ont pas besoin d'émettre d'interactions.

Durant chacune des expérimentations, nous avons relevé la moyenne du nombre d'interactions émises par chaque agent rapportée sur la durée complète de la simulation. Ce chiffre permet de vérifier que le nombre d'interactions émises ne diffère pas trop d'une expérimentation à l'autre. Si cette moyenne était trop différente d'une expérimentation à l'autre (pour un même comportement modélisé), cela aurait signifié que les différents systèmes (approche SAMP et approche classique) n'exécutaient pas le même comportement. Nos tests en auraient été faussés.

Pour les expérimentations avec le SMA proies-prédateurs complet, cette moyenne oscille entre 0.13 et 0.15. Pour les expérimentations avec les moutons aveugles, cette moyenne est de 0 pour chaque expérimentation. En effet, le nombre d'agents émettant des inter-

1. Ces deux valeurs du nombre d'agents ont été obtenues une fois que nous avons créé les scènes contenant les entités 3D des simulations

2. La population se stabilisant aux alentours des 4000 frames, nous avons fixé la fin d'exécution à 5000 frames

actions est minime. Chaque mouton n'émet plus d'interactions que vers les agents loups à proximité. Dans le SMA proies-prédateurs complet, les moutons et les carrés de sol émettent des interactions vers les moutons qui leur sont proches (pour respectivement la reproduction entre les moutons et pour l'alimentation des moutons).

Pour rappel, le système proies-prédateurs que nous utilisons pour nos expérimentations est composé de loups, de moutons et de carrés d'herbes. Chaque mouton se déplace aléatoirement. Lorsqu'il a faim, il se met en recherche d'un carré d'herbe qui n'a pas encore été mangé. Une fois qu'il en a trouvé un, il se dirige vers lui et le mange. Ensuite, il a un pourcentage de chance de chercher à se reproduire. S'il ne cherche pas à se reproduire, il va retourner dans sa phase de déplacement aléatoire. S'il veut se reproduire, il va chercher un mouton, se rapprocher de lui et se reproduire avec puis retourner dans sa phase de déplacement aléatoire.

Les loups ont un comportement similaire à ceci près qu'ils mangent des moutons et non de l'herbe et se reproduisent avec des loups. Lorsqu'un mouton est mangé par un loup, il meurt.

Lorsqu'ils sont mangés, les carrés d'herbes changent de représentation visuelle et deviennent non comestibles pendant une certaine période. Lorsque cette période est passée, ils redeviennent comestibles.

Chaque expérimentation est réalisée 50 fois avec, pour chaque exécution, une graine différente pour l'aléatoire. Chaque expérimentation est exécutée 50 fois avec les mêmes graines que les autres.

8.3/ EXPÉRIMENTATIONS AVEC LE SMA PROIES-PRÉDATEURS COMPLET

Nous avons commencé nos expérimentations en exécutant nos tests sur des SMA reproduisant l'exemple proies-prédateurs que nous avons précédemment décrit. Comme nous l'avons expliqué dans la section 8.2, nous avons réalisé 4 expérimentations pour ce proies-prédateurs. Deux pour chaque système d'interactions chacun dans deux configurations de SMA différentes.

Nos résultats sont visibles dans les graphes des figures 8.1 et 8.2.

Sur ces résultats, nous constatons que SAMP est constamment plus efficace que l'approche classique. Nous pouvons aussi constater que plus le nombre d'agents grandi plus l'écart devient important.

8.4/ EXPÉRIMENTATIONS AVEC DES MOUTONS AVEUGLES

Dans le but de tester l'impact des interactions sur l'approche SAMP et sur son efficacité par rapport à l'approche classique nous avons réalisé de nouvelles expérimentations en modifiant le comportement des moutons. Pour ce faire, nous avons rendu les moutons aveugles. C'est-à-dire que nous leur avons enlevé leur compétence de vue. Ainsi, ils n'exécutent plus de scan de leur environnement pour ce qui concerne leur vision. Nous avons exécuté les mêmes expérimentations que dans la section 8.3. Les résultats sont

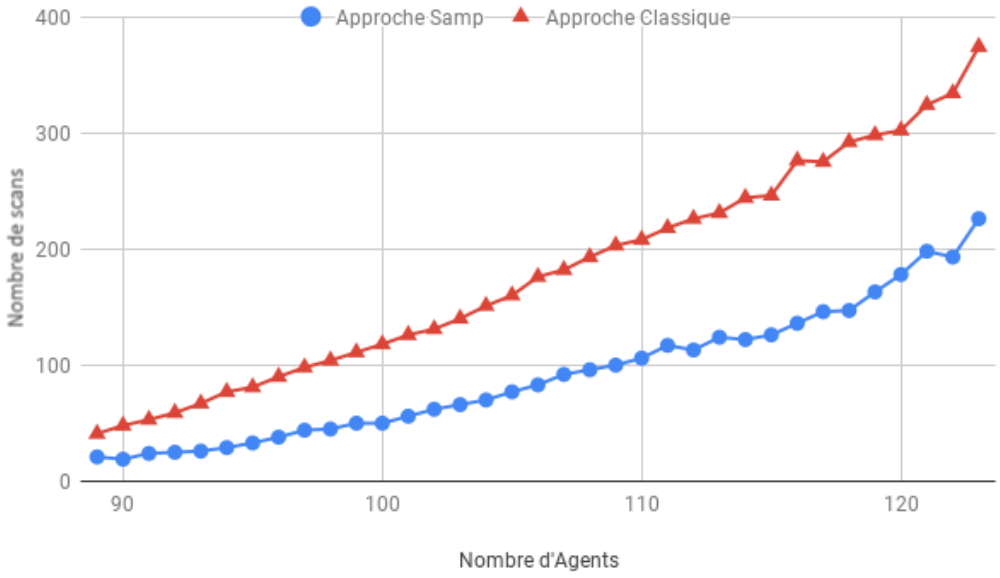

FIGURE 8.1 – Nombre moyen de scans par frame en fonction du nombre d'agents pour proies-prédateurs complet (nombre d'agents réduit).

visibles dans les figures 8.3 et 8.4

On constate avec ces figures que les tendances restent les mêmes. L'approche SAMP reste plus efficace que l'approche classique et tend à devenir de plus en plus efficace plus le nombre d'agents augmente.

En comparant les résultats pour l'approche SAMP dans les deux expérimentations (complète et avec les moutons aveugles), on constate que le nombre de scans réalisés par chaque agent est légèrement plus faible lorsque les moutons sont aveugles. Dans le cas où les moutons sont aveugles, lorsque ceux-ci sont actifs, ils ne scannent plus les carrés d'herbes, mais vont tout de même scanner les loups. En effet, même si les moutons ne peuvent pas voir les loups, les loups doivent tout de même être notifiés de leur présence. C'est pourquoi le nombre de scans réalisés par les agents est légèrement plus faible bien que les agents moutons ne soient plus intéressés par aucune interaction.

Ce qui a le plus d'impact sur notre approche, en termes de nombre de scans effectués, ce n'est pas la complexité des agents qui composent le SMA mais leur nombre.

8.5/ CONSOMMATION DE MÉMOIRE

Le but de notre approche de gestion des interactions est de diminuer le nombre de scans qu'effectuent les agents. Nous avons constaté que les agents de SAMP effectuent moins de scans de leur environnement qu'avec une approche classique. Cependant, cette efficacité a un prix.

Le principe de l'approche de SAMP repose sur le fait que les agents ne scannent leur environnement que lorsqu'ils sont actifs. Lorsqu'ils scannent leur environnement, les agents enregistrent dans un tableau les agents intéressés par leurs interactions et s'enregistrent

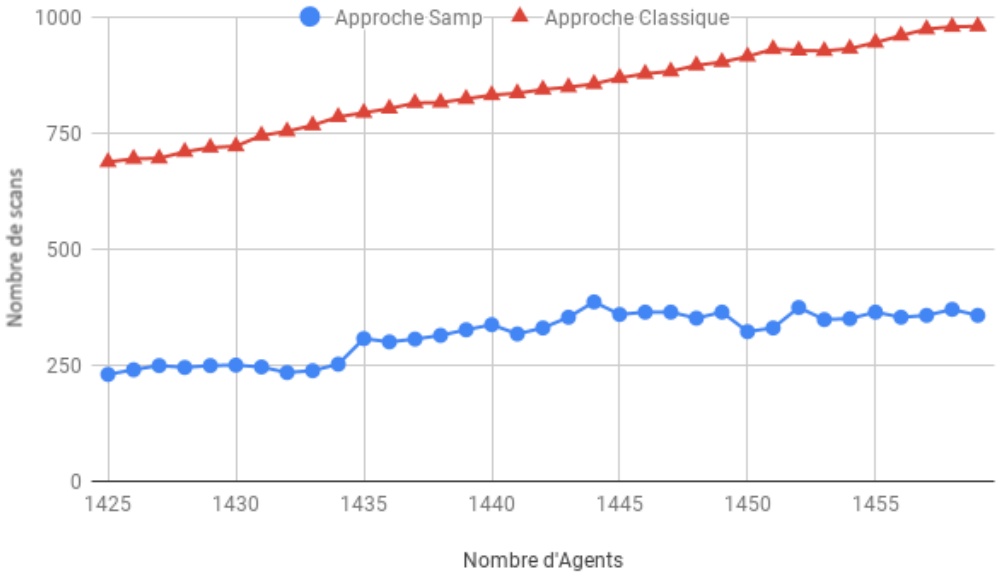

FIGURE 8.2 – Nombre moyen de scans par frame en fonction du nombre d'agents pour proies-prédateurs complet (grand nombre d'agents).

dans un tableau possédé par les agents dont les interactions les intéressent.

Par rapport à l'approche classique, chaque agent possède deux tableaux supplémentaires. Le premier contient une liste des agents émettant des interactions vers le propriétaire du tableau. Le second contient une liste des agents vers lesquels l'agent propriétaire du tableau émet des interactions. Chaque élément de ces tableaux est composé de 2 variables : les éléments du premier tableau sont composés d'un pointeur vers l'agent qui émet l'interaction et d'un pointeur vers cette interaction. Les éléments du second tableau sont composés d'un pointeur vers l'agent qui reçoit l'interaction et d'un pointeur vers cette interaction.

Le coût supplémentaire est donc, pour chaque interaction émise, de $4 pointeurs * 8 octets = 32 octets$. Lors de nos expérimentations, le plus grand nombre d'interactions que nous avons obtenu était de 25003 ce qui donne un surcout total d'environ 800 kilo-octets. Ce surcoût semble minime comparé aux avantages apportés par cette approche. Mais il nous semble intéressant pour le futur de nos travaux de diminuer ce surcoût afin d'améliorer encore notre approche.

8.6/ MODÉLISATION D'UN JEU 2D

Afin de s'assurer que SAMP est adapté à plusieurs types de jeu vidéo, nous avons modélisé un autre SMA. Ce SMA modélise un jeu en deux dimensions (2D) vue du dessus. Ce SMA est composé d'un agent personnage (contrôlable par l'utilisateur), d'ennemis et d'éléments de décors. Dans la suite de cette section, nous décrivons plus en détail ce SMA puis nous expliquons comment nous avons réussi à le modéliser.

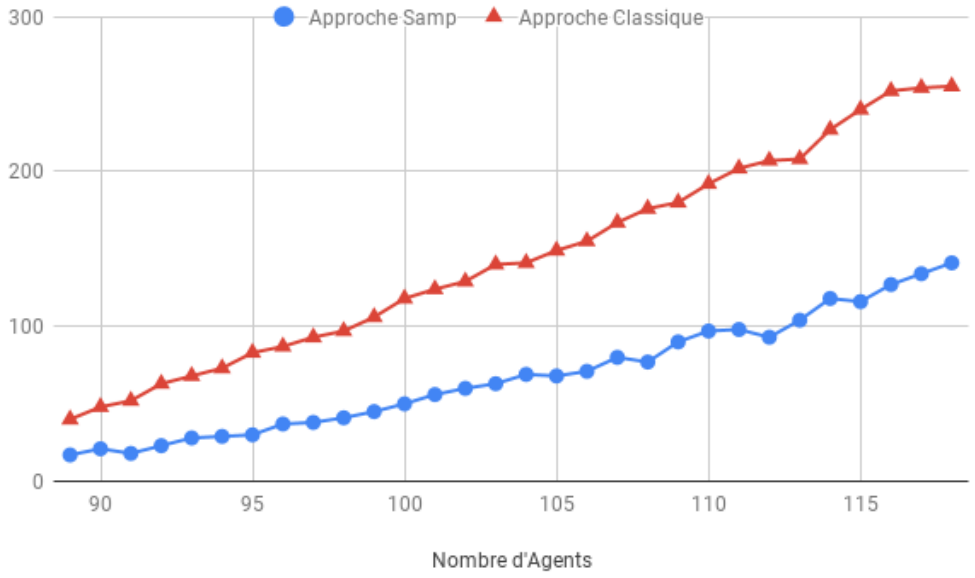

FIGURE 8.3 – Nombre moyen de scans par frame en fonction du nombre d'agents pour proies-prédateurs (moutons aveugles et nombre d'agents réduit).

8.6.1/ DESCRIPTION DU SYSTÈME

Il existe deux possibilités de définition pour un jeu 2D :

— D'un point de vue rendu, un jeu 2D est un jeu dont les ressources sont elles-mêmes en 2D et dont le rendu se fait sur un plan ;

— D'un point de vue *Gameplay*, un jeu 2D est un jeu dans lequel les entités sont positionnées sur un plan.

Le jeu que nous avons développé est un jeu 2D d'un point de vue rendu et d'un point de vue *gameplay*. Nous avons choisi de faire un rendu 2D, car nous avons déjà prouvé qu'un jeu en 3D était possible avec l'exemple proies-prédateurs.

L'environnement est découpé sous la forme d'une grille où chaque agent est positionné en fonction de la case sur laquelle il se trouve. Il existe plusieurs types d'agents :

— Le personnage qui est l'agent que les utilisateurs contrôlent. Il peut se déplacer, ramasser des objets et attaquer ses ennemis. Il possède des points de vie. A chaque fois qu'un ennemi le touche, il perd de la vie. Pour attaquer un ennemi, l'utilisateur doit exécuter l'action attaque en étant sur une case adjacente à un ennemi et en étant tourné dans sa direction ;

— Les ennemis sont des agents qui se meuvent à la recherche du personnage afin de l'attaquer. Dès qu'un ennemi voit le personnage, il le suit jusqu'à pouvoir l'attaquer ;

— Les pièces sont des agents que le personnage doit ramasser afin de gagner la partie. Dès que le personnage se trouve sur une case avec une pièce, il la ramasse automatiquement ;

— Les potions sont des agents qui rendent le personnage invulnérable.

De plus, les ennemis et le personnage possèdent une valeur de direction indiquant le sens dans lequel ils regardent. Cette direction peut être *droite*, *gauche*, *haut* ou *bas*.

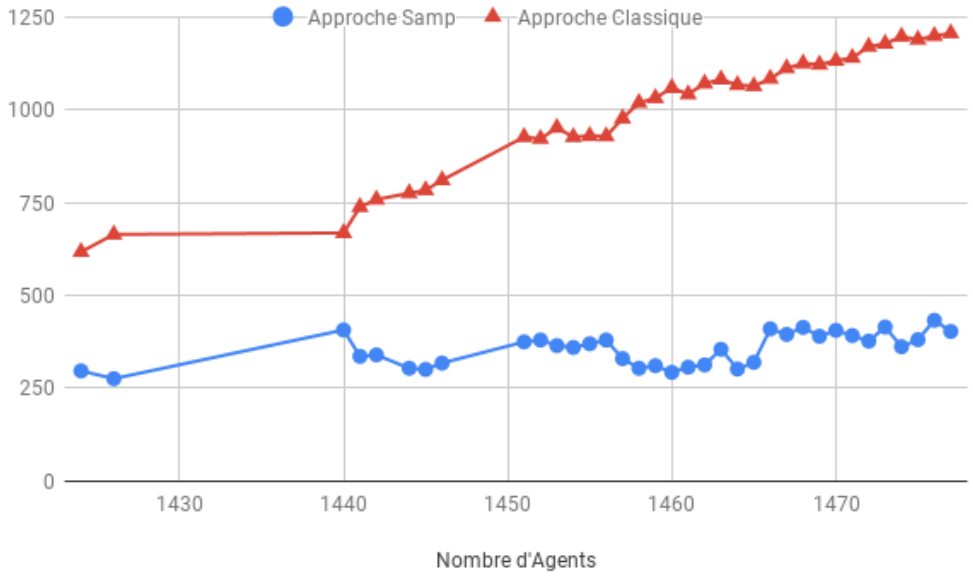

FIGURE 8.4 – Nombre moyen de scans par frame en fonction du nombre d'agents pour proies-prédateurs (moutons aveugles et grand nombre d'agents).

Les ennemis et le personnage ne peuvent se déplacer que dans ces quatre directions. Lorsqu'un ennemi arrive sur la même case que le personnage, le personnage perd un point de vie et l'ennemi est tué.

Les déplacements du personnage se font en utilisant les touches fléchées du clavier et l'attaque se fait en appuyant sur la touche *espace*. Un appui sur une touche fléchée aura deux effets selon la direction dans laquelle le personnage regarde. S'il regarde dans la même direction que la flèche appuyée, il se déplacera. S'il regarde dans une direction opposée, il devra d'abord s'orienter dans la bonne direction avant de se déplacer. L'orientation du personnage prend du temps, tout comme le temps d'attente entre deux attaques, et il s'agit d'éléments du gameplay. Lorsque le personnage est invulnérable, les ennemis qui le voient fuient.

8.6.2/ L'AGENT PERSONNAGE

L'agent personnage est l'agent que l'utilisateur contrôle en utilisant les touches fléchées. Il possède deux propriétés : une propriété *point de vie* déterminant combien de fois il peut subir des dégâts de la part d'un ennemi et une propriété case déterminant la case sur laquelle il se trouve.

Il possède aussi des compétences. Une compétence attaque permettant d'attaquer les ennemis présents (émettre une interaction en direction des ennemis) dans l'environnement et une compétence *être attaqué* qui lui permet de recevoir une interaction de la part des ennemis qui le touche.

Il y a 10 modèles qui gèrent le comportement de l'agent personnage :

— Un modèle *comportement* modélisant le comportement global de l'agent ;

— Un modèle *état* appelé *MainCharacter* qui modélise le comportement principale de

l'agent ;

— Un modèle *état* appelé *idle* qui modélise le comportement lorsque l'agent est à l'arrêt ;

— Un modèle *état* appelé *dead* qui modélise le comportement lorsque l'agent est mort ;

— Un modèle *état* appelé *moveTo* qui modélise le comportement de l'agent lorsqu'il se déplace ;

— Un modèle *état* appelé *attack* qui modélise le comportement de l'agent lorsqu'il attaque ;

— Un modèle *état* appelé *attacked* qui modélise le comportement de l'agent lorsqu'il est attaqué ;

— Un modèle *événement* appelé *actiontriggered* qui modélise l'événement déclenché lorsqu'une touche du clavier est appuyée ;

— Un modèle *événement* appelé *OnContact* qui modélise l'événement déclenché lorsqu'un autre agent entre en contact avec le personnage ;

— Un modèle *altération* qui modélise les altérations que subit le personnage (perte de point de vie, changement de statistique, ...).

8.6.3/ LES AGENTS ENNEMIS

Les agents ennemis sont des agents capables de se mouvoir dans l'environnement. Ils possèdent deux propriétés :

— Une propriété *Case* permettant de connaître la case sur laquelle ils se situent ;

— Une propriété *CasePersonnage* permettant de stocker la dernière position du personnage qu'il connaît.

Les ennemis se déplacent aléatoirement sur le plateau. Lorsqu'ils voient l'agent personnage, ils enregistrent la case sur laquelle ils le voient comme étant une destination. Lorsqu'ils atteignent cette destination (ce qui signifie qu'ils ne voient plus le personnage), ils recommencent à se déplacer aléatoirement. Lorsqu'un ennemi atteint la même case que le personnage, il meurt. Lorsque le personnage attaque un ennemi, il meurt aussi.

Pour se déplacer, ils exécutent le même modèle de déplacement que le personnage (MoveTo décrit en 8.6.8).

8.6.4/ LES AGENTS MURS

Les agents murs servent à délimiter les zones dans lesquelles les autres agents ne peuvent pas aller. Ces agents ne possèdent qu'une seule propriété : une propriété case permettant de déterminer la case sur laquelle ils se trouvent. Ils ne possèdent aucune compétence.

Le seul comportement que les agents case exécutent est, au moment de leur initialisation, le calcul de la case sur laquelle ils se trouvent en fonction de leurs positions dans l'environnement. La figure 8.5 expose la modélisation du calcul de la valeur de la propriété case des murs.

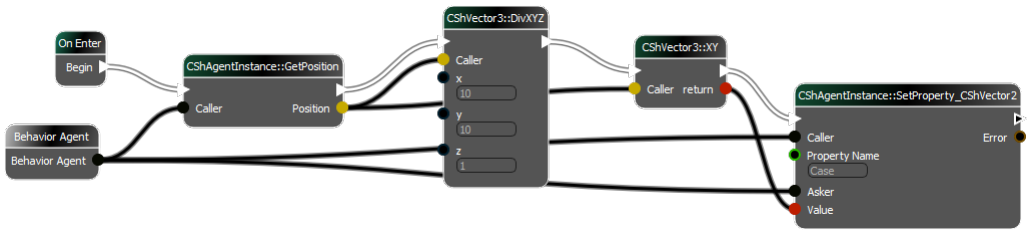

FIGURE 8.5 – Modèle *Comportement* calculant la valeur de la case de chaque mur

Nous considérons que chaque case de l'environnement est un carré de 10 de côté. Ainsi, la case sur laquelle se trouve chaque agent est sa position dans l'environnement divisé par 10.

8.6.5/ LES AGENTS POTION

Un agent potion est un agent qui, lorsque le personnage entre sur sa case, disparaît et redonne un point de vie au personnage. Un agent potion possède une propriété case permettant à l'agent de connaître sa position sur la grille de l'environnement.

Les agents potions ont un comportement très simple. Ils attendent que le personnage leur transmette une interaction de contact. A ce moment-là, les potions disparaissent. Lorsque le personnage entre en contact avec une potion il regagne un point de vie et émet une interaction à destination de l'agent potion qu'il vient de "ramasser".

8.6.6/ LA DÉTECTION DES ENTRÉES UTILISATEURS

Rappelons que *Shine Engine* permet la gestion des entrées utilisateurs quelle que soit la plateforme sur laquelle l'application est exécutée. Pour ce faire, Shine Engine utilise ce que nous appelons des *actions*. Chaque action est paramétrée avec, pour chaque plateforme, le périphérique utilisé (manette, clavier, souris, etc), le capteur utilisé (nom du bouton, de la touche du clavier, de la gâchette, etc) ainsi que l'état de ce capteur qui activera l'action (pressé, relâché, etc). Lorsque l'on veut savoir si une action est activée, on appelle alors la méthode *HasTriggeredAction* de la classe *ShUser*.

La figure 8.6 expose le modèle *événement* qui permet de modéliser la condition de déclenchement de l'événement détectant l'activation d'une action de *Shine Engine*. Ce modèle est alors utilisé, pour l'agent personnage, dans le modèle *MainCharacter*. La figure 8.7 fait un zoom sur le modèle *MainCharacter* pour n'exposer que les nœuds permettant de passer de l'état *idle* (l'agent est immobile) à l'état *moveTo* (l'agent se déplace) lorsque l'on appuie sur une touche fléchée. Lorsqu'un des événements est déclenché, on créé un vecteur de direction paramétré en fonction de l'événements déclenché. C'est ce vecteur direction qui sera passé en entrée du modèle *moveTo*

8.6.7/ LA DÉTECTION DES COLLISIONS

L'agent personnage et les ennemis sont sensibles aux collisions. Ils ne peuvent pas se déplacer sur une case sur laquelle se trouve un mur. Lorsqu'un ennemi entre dans la case

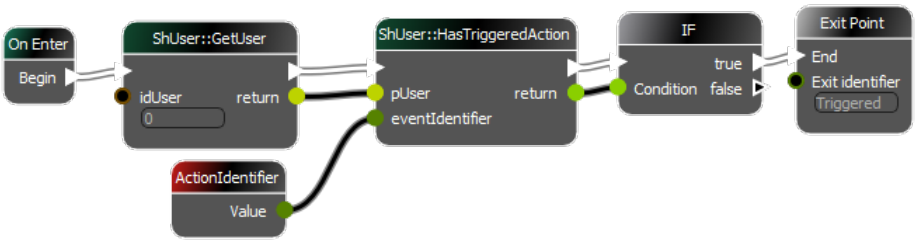

FIGURE 8.6 – Modèle *événement* détectant le déclenchement d'une entrée utilisateur

où se trouve le personnage, ce dernier perd un point de vie et l'ennemi est tué. Lorsque le personnage attaque en direction d'une case où se trouve un ennemi, ce dernier est tué et lorsque le personnage se déplace sur une case sur laquelle se trouve une potion, il la ramasse et ses effets sont appliqués au personnage.

Nous avons envisagé deux possibilités pour la gestion des positionnements et des collisions :

— La première solution consiste en la création d'agents *case* qui auraient pu contenir une liste d'agents correspondant aux agents présents sur chaque case ;

— La seconde solution consiste en la recherche dans l'environnement des agents à proximité et de récupérer le ou les agents présents sur la case qui nous intéresse.

Ces deux solutions nous semblaient bonnes à implémenter, mais dans l'optique de montrer les capacités de modélisation de SAMP, il nous a semblé plus opportun d'utiliser la seconde solution qui fait plus intervenir les interactions entre les agents et l'environnement.

Ainsi, pour détecter si le personnage entre en collision avec un autre agent, il faut chercher les agents présents dans son environnement proche et récupérer la case sur laquelle ils se trouvent afin de savoir si un agent se trouve sur la même case que le personnage.

La figure 8.8 présente un zoom sur la modélisation de la détection de collisions. Il s'agit dans cette figure de détecter si le personnage va entrer en collision avec un mur, et dans ce cas, arrêter son déplacement.

Dans ce modèle, on récupère tous les agents proches de l'agent personnage (à une distance de 11) et, pour chaque agent récupéré, on vérifie la valeur de sa case. Si elle correspond à la valeur de la case où l'on veut aller et que le type de l'agent est *Wall* cela signifie qu'il y a un mur là où l'agent veut se déplacer. Le booléen *CanMove* est alors assigné à la valeur *faux*. Lorsque l'on sort de la boucle qui parcourait tous les agents proches, si le booléen *CanMove* est à *vrai* le personnage se déplace, sinon, il retourne dans l'état *idle* (Non présent dans le zoom fait pour la figure 8.8).

8.6.8/ UN MODÈLE RÉUTILISABLE : *MoveTo*

Le modèle permettant le déplacement des agents est le même qu'il s'agisse de l'agent personnage ou des agents ennemis. La figure 8.9 expose le modèle permettant le déplacement des agents dans leur environnement. Ce modèle prend en entrée un vecteur de direction normalisé. A l'aide de ce vecteur direction, le modèle calcule, dans la phase

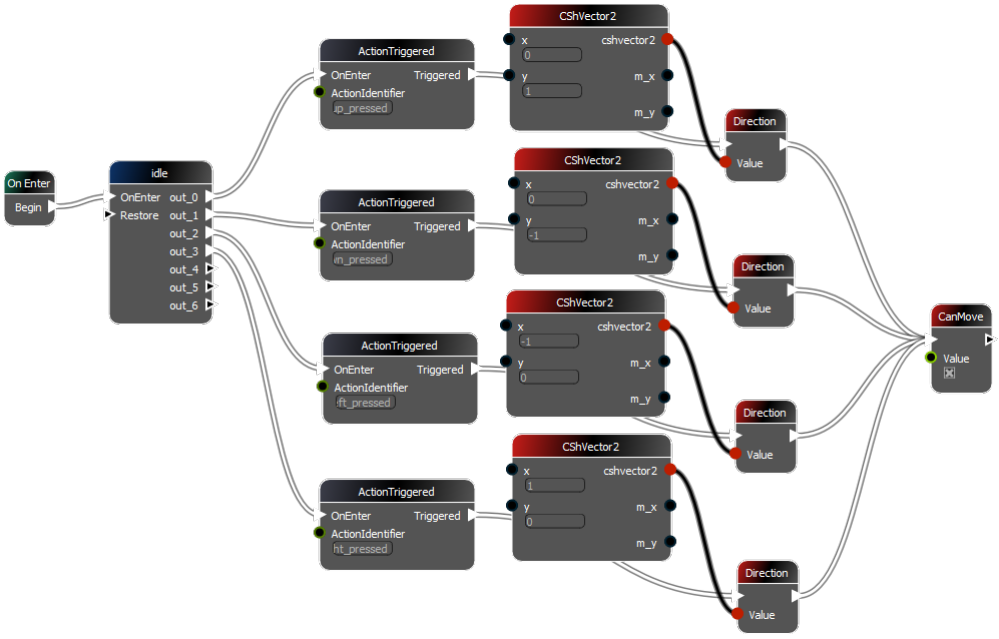

FIGURE 8.7 – Zoom sur l'utilisation des événement *actionTriggered* dans le Modèle *Main-character*

OnEnter, la case qui sera la destination du déplacement. Une fois la case destination calculée, le modèle paramètre la direction et la vitesse de l'agent pour le faire se déplacer.

Dans la phase *PostUpdate*, le modèle vérifie si l'agent a atteint la destination visée. Si c'est le cas, il déclenche un événement interne indiquant qu'il a atteint la destination ciblée.

8.6.9/ LE MODÈLE D'ALTÉRATION DU PERSONNAGE

Dans SAMP, les modèles d'altérations permettent de définir comment les agents réagissent aux événements qui ne les font pas changer d'état. Dans le cas de ce jeu 2D, le modèle altération permet de gérer un événement : Lorsqu'un ennemi entre dans la case sur laquelle se trouve le personnage. Dans ce cas-là, le personnage perd un point de vie et l'ennemi est tué.

La figure 8.10 expose le modèle altération du personnage. Dans ce modèle, on voit que pour tuer l'ennemi, le personnage lui envoie une interaction *attack*. En effet, dans son comportement, lorsqu'un agent ennemi reçoit une interaction *attack*, il change d'état pour entrer dans un état de mort.

8.7/ SAMP DANS D'AUTRES PROJETS

SAMP est utilisable dans de nombreux projets différents. Cela peut être dans une simulation de type proies-prédateurs, un jeu vidéo en 2D, mais aussi dans des projets de jeu 3D ou pour la gestion d'interfaces utilisateur.

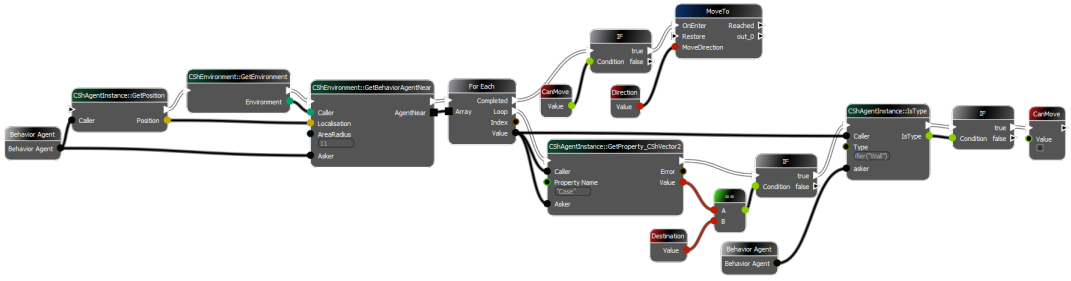

FIGURE 8.8 – Zoom sur la détection de collision entre le personnage et les murs

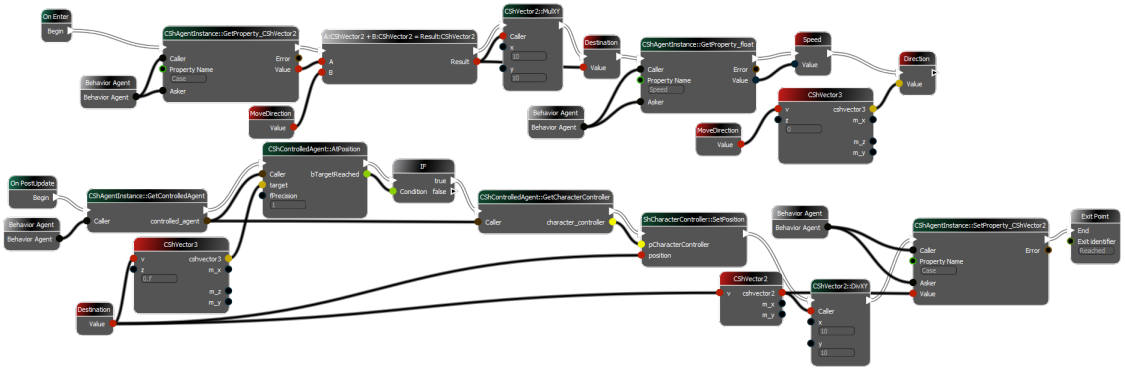

FIGURE 8.9 – Zoom sur la détection de collision entre le personnage et les murs

8.7.1/ SILVA NUMERICA

Silva Numerica [3] est un projet mené en collaboration entre différentes structures dont de nombreux instituts et laboratoires de recherches : AMValor, l'Université de Bourgogne Franche-Comté, le Laboratoire d'Etude de l'Apprentissage et du Développement (LEAD), Eduter Recherche (Agrosup Dijon). L'objectif de Silva Numerica est de développer un logiciel de simulation de gestion de forêt à l'attention des professionnels de la filière du bois, mais aussi à l'attention d'élèves de collège, lycée, Bts et filière professionnalisante.

Dans ce logiciel, l'utilisateur contrôle une entité pouvant observer une parcelle d'une forêt. Le point de vue de l'utilisateur peut être un point de vue personnage (la caméra est située à la hauteur du regard de l'entité) ou être un point de vue placé au-dessus de la forêt.

Dans ce projet, SAMP est utilisé pour modéliser et gérer les comportements de l'entité contrôlée par l'utilisateur, de ses actions sur son environnement et des arbres et autres végétaux dans l'environnement.

Dans ce projet, on utilise le modèle de détection des entrées utilisateurs, déjà utilisé dans le projet de jeu 2D décrit précédemment. Le modèle permettant le déplacement des agents que nous avons créé dans le système proie-prédateur est aussi utilisé dans ce projet pour le déplacement du personnage que nous contrôlons.

3. http://www.silvanumerica.net/

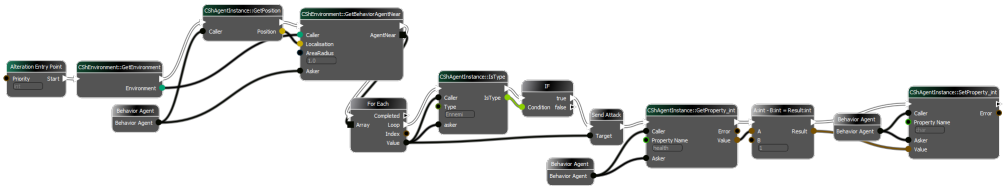

FIGURE 8.10 – Modèle *altération* du personnage

8.7.2/ UN LECTEUR DE FLUX RSS

SAMP permet de faire des jeux et des simulations, mais il permet aussi de modéliser le comportement d'interface utilisateur comme un menu de jeu par exemple.

Nous avons alors modélisé le comportement d'un lecteur de flux Rss. Il s'agit d'une interface utilisateur récupérant le flux Rss à partir d'un lien internet, récupère les données du flux Rss et les affiches. Cette interface se découpe en deux parties : une première partie permet d'afficher la liste complète des éléments récupérés sur le flux Rss et d'afficher leurs titres et résumés. Une deuxième partie permet l'affichage du contenu d'un élément qui est sélectionné.

En utilisant SAMP, il y a un agent qui gère la récupération des données du flux. Cet agent possède une propriété indiquant l'adresse du flux Rss. Ensuite, chaque entrée du flux Rss est affichée dans l'interface. Chacune de ces entrées est un élément d'interface sur lequel les utilisateurs peuvent cliquer et qui affiche le titre et le résumé d'un élément du flux Rss. Chacun de ces éléments est associé à un agent SAMP qui réagit lorsque l'utilisateur clique sur l'élément graphique ce qui a pour conséquence d'afficher en plein écran le contenu de cette entrée. Le contenu affiché en plein écran associé à un autre agent.

Chaque bouton présent dans l'interface est aussi associé à un agent : le bouton *actualiser* permettant de récupérer les données du flux Rss et le bouton *retour* permettant de passer de l'affichage plein écran d'une entrée du flux à l'affichage de la liste des entrées.

La possibilité de modéliser le comportement d'interfaces utilisateur permet d'offrir aux utilisateur débutants la possibilité de créer les menus de leur jeu sans avoir à utiliser un autre système que SAMP. Ajoutée à la possibilité d'importer des modèles d'autres projets, la création des interfaces utilisateur peut être faite sous forme de différents modules (menu vertical, menu horizontal, ...) par des utilisateurs avancés et utilisée par des utilisateurs débutants n'ayant qu'à associer les modules entre eux afin d'obtenir le comportement voulu.

La figure 8.11 expose le comportement de l'agent listant les éléments du flux Rss. Cet agent débute en étant dans un état où il affiche la liste éléments du flux. Lorsqu'un clic est réalisé sur un élément de la liste, il récupère cet événement et entre dans un état d'affichage d'un élément. Dans cet état, il rend invisible l'objet de l'interface qui lui est associé et émet une interaction vers l'agent gérant l'affichage d'un élément du flux en plein écran. Cette interaction transmet le pointeur sur l'agent dont l'élément associé vient d'être cliqué.

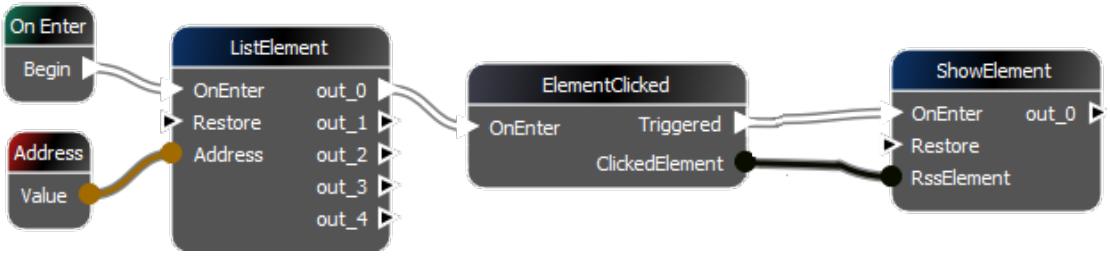

FIGURE 8.11 – Modèle *Comportement* de l'agent gérant la liste des éléments du Rss

8.8/ BILAN DES EXPÉRIMENTATIONS

Dans cette section, nous avons exposé les résultats de nos expérimentations. Nous avons tout d'abord validé les résultats théoriques obtenus dans le chapitre 7.

L'approche SAMP visant à réduire le nombre de scans de l'environnement que chaque agent réalise est plus efficace qu'une approche classique. Il faut tout de même noter que cela s'applique dans le cas où les agents cherchent à connaître l'état de leur environnement à chaque frame. C'est notamment le cas dans le domaine des jeux vidéo. Chaque agent pouvant interagir avec le reste du monde doit connaître son environnement à chaque instant afin de pouvoir réagir aux perturbations qui pourraient impacter son comportement.

L'approche SAMP apporte cependant une légère augmentation de la consommation de mémoire par rapport à l'approche classique. De l'ordre de 32 octets par interaction émise, ce coût peut nous sembler dérisoire par rapport à la consommation en mémoire des jeux vidéo actuels. En nous référant à une valeur de 5 giga-octets utilisés pour un jeu vidéo [4] et en utilisant la valeur de surconsommation donnée dans la section 8.5 qui est de 781 kilo-octets, nous obtenons un surcoût de 0.015% de l'utilisation de la mémoire. De plus, l'exemple que nous avons utilisé implique que les agents ont, à chaque instant, besoin de connaître l'état de leur environnement. Sans cette contrainte, il est envisageable en approche classique que les agents, même actifs, ne scannent leur environnement que lorsqu'ils en ont besoin, réduisant ainsi le nombre de scans exécutés. Dans SAMP, cette optimisation ne serait pas possible, car chaque agent actif, même s'il n'a besoin d'aucune interaction, doit scanner son environnement pour pouvoir éventuellement notifier les agents passifs. Cependant, dans un jeu vidéo il est rare que les agents actifs n'aient pas besoin de scanner leur environnement notamment pour détecter des obstacles sur leurs routes.

Dans la section 8.6 nous avons modélisé un jeu 2D afin de tester les possibilités offertes par SAMP. Ce jeu est très différent de l'exemple proies-prédateurs, mais utilise cependant des éléments importés de la modélisation du proies-prédateurs. Cela montre que le principe de nœuds exportables fonctionne correctement et permet aux novices en développement d'utiliser ce que d'autres utilisateurs ont développé.

Enfin, les sections 8.7.1 et 8.7.2 permettent de confirmer le fait que SAMP puisse permettre de modéliser des comportements très différents, éloignés des comportements d'agents dans un jeu vidéo. La possibilité de modéliser le comportement du système à

4. Les configurations minimales pour la mémoire des jeux vidéo actuels avoisinent souvent les 5 giga-octets et approchent de plus en plus des 10 giga-octets.

travers les interfaces utilisateur permet d'ouvrir SAMP à la création d'applications et logiciels par un maximum d'utilisateurs, quel que soit leur niveau en développement logiciel.

IV

CONCLUSION ET PERSPECTIVES

9

CONCLUSION

Nous rappelons ici les problématiques qui ont été les nôtres ainsi que les objectifs que nous avons fixés pour y répondre. Nous revenons ensuite sur les contributions que nous avons développées pour atteindre ces objectifs. Nous continuons par une discussion autour des perspectives d'évolution de SAMP. Ce chapitre se termine par une conclusion plus personnelle sur l'aventure qu'est la préparation d'un doctorat.

9.1/ LE CONTEXTE

Le jeu vidéo et les SMA sont deux domaines qui sont complémentaires.

Le domaine des SMA est le terrain d'un grand nombre de travaux de recherche visant, entre autres, à simuler des systèmes distribués réels ou virtuels. Dans notre cas, les agents peuvent tout à fait représenter les entités d'un jeu vidéo. Ces travaux apportent une expertise en termes d'interactions, de validations, mais aussi de prises de décisions, de coopérations et de négociations, autant de concepts qui peuvent être utiles aux jeux vidéo.

Ces jeux vidéo apportent aussi une expertise dans les domaines de la simulation des concepts de la physique, des algorithmes de rendu 2d et 3d pour l'affichage des simulations et des interfaces utilisateur (tant au niveau de l'ergonomie des affichages que de l'utilisation des périphériques d'entrée).

Nous nous plaçons dans un contexte où les SMA sont utilisés comme paradigme de programmation des jeux vidéos. Mais la fusion de ces deux domaines est accompagnée de l'apparition de différents verrous : comment assurer la performance d'exécution afin de respecter le concept de *frame* présent dans le domaine du jeu vidéo, sachant que les SMA ne sont pas toujours optimisés dans ce domaine ? Comment intégrer des outils externes tels que les moteurs physiques ? Nous avons constaté que les plateformes de modélisation de SMA existantes ne sont pas forcément adaptées au développement de jeux vidéo, tout du moins dans le respect de ces contraintes.

Au-delà des difficultés liées à la fusion de ces deux domaines, il existe un problème commun qui concerne la traduction des besoins entre la personne désirant modéliser un système et le développeur qui va le modéliser. Dans le domaine des SMA, lorsqu'un scientifique, une collectivité ou un industriel expose ses besoins à un développeur, il y a un risque que cette personne les exprime mal ou/et que le développeur les comprenne mal. Dans le domaine du jeu vidéo, le problème est le même lorsqu'un *Game Designer*,

un animateur 3D ou un *Level Designer* expose ses besoins à un développeur. Certains outils existent pour permettre aux utilisateurs voulant modéliser un SMA ou un jeu de le faire eux-mêmes. Mais aucun de ces outils, à notre connaissance, ne permet le développement de jeux vidéo en utilisant le paradigme multi-agents.

Forts de cette analyse, nous avons extrait 3 objectifs qui ont guidé nos travaux durant cette thèse. Nous avons eu pour objectifs le développement d'une plateforme **de modélisation, de génération et d'exécution de jeux vidéo basé sur le paradigme multi-agents accessible à un grand nombre d'utilisateurs** afin de pallier les problèmes de traduction entre les différents acteurs de la modélisation de SMA. Ce logiciel devait **intégrer des concepts et spécificités propres au domaine des jeux vidéo et au domaine des multi-agents** afin d'allier leurs optimisations. Il devait aussi **s'approcher d'une exécution en temps réel** afin d'offrir une fluidité et une immersion de bonne qualité nécessaires au plaisir de jeu.

9.2/ NOS CONTRIBUTIONS

Afin de répondre à ces objectifs, nous avons développé *Shine Agent Modeling Platform (*SAMP*)*. Cette plateforme permet d'unifier les domaines du jeu vidéo et des SMA. Nous y avons intégré les concepts du paradigme multi-agents (interactions, perceptions, compétences, environnement, ...).

Afin de se rapprocher au plus près d'un fonctionnement en temps réel, nos travaux ont été guidés par deux impératifs : le respect du concept de *frame* propre aux jeux vidéo et la présence de capteurs (les boutons des périphériques d'entrées) dont les informations doivent être transmises en temps réel au système.

Plusieurs travaux de recherche dans le domaine des SMA visent à optimiser le fonctionnement des SMA pour s'approcher d'une exécution en temps réel. Cependant, la plupart de ces recherches sont incompatibles avec le domaine du jeu vidéo. Les recherches sur l'amélioration des architectures matérielles ne pouvaient pas être intégrées du fait de l'hétérogénéité des systèmes sur lesquels les jeux sont exécutés (chaque joueur possède un pc différent et *Shine Engine* est entièrement multi-plateforme). Les travaux visant à exécuter les SMA sur les *GPU* sont aussi incompatibles car, dans le domaine des jeux vidéo, les ressources des *GPU* sont dédiées aux calculs de rendu et de physique.

C'est pourquoi nous avons décidé de concentrer nos recherches sur l'amélioration des interactions entre les agents, gourmandes, par leur nombre, en temps d'exécution. Notre approche, basée sur des agents actifs et passifs, permet une réduction du nombre de recherches effectuées dans l'environnement par les agents. Nos expérimentations ont prouvé que cette approche est efficace dans bien des contextes.

En sus d'améliorer les interactions entre les agents, SAMP a pour objectif d'être accessible au plus grand nombre d'utilisateurs. Pour ce faire, il met à disposition un éditeur graphique permettant la modélisation de SMA, du paramétrage des agents à la modélisation de leur comportement. L'éditeur graphique de SAMP ne requiert aucune connaissance de langage de développement textuel et offre ainsi une accessibilité simplifiée à un grand nombre d'utilisateurs. La modélisation des comportements se fait à l'aide d'un système de graphes de nœuds et les paramétrages des agents, interactions et compétences se font par le biais de fenêtres graphiques.

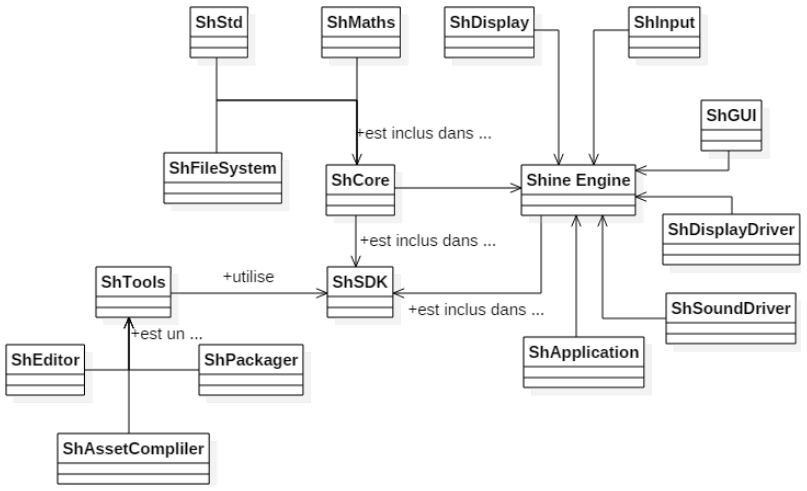

FIGURE 9.1 – Diagramme de classes de Shine Engine

Afin d'assurer un bon fonctionnement, tant de la partie modélisation des SMA que de l'exécution du modèle, SAMP est composé de 3 modules :

— SAMP-M est un méta-modèle permettant de définir les éléments composant SAMP (interactions, compétences, agents, modèles comportementaux, ...). SAMP-M permet aussi de définir les règles liant chacun de ces éléments aux autres ;

— SAMP-E est l'éditeur graphique de SAMP permettant de paramétrer les projections du méta-modèle : paramétrages des compétences, des interactions et des agents, modélisation des comportements, ... ;

— SAMP-X qui réalise la génération, la compilation et l'exécution des SMA modélisés.

SAMP intègre, en plus des concepts agent, des outils propres au domaine des jeux vidéo. La gestion des principes physiques (gravité, collisions, calculs des distances, propagation du son, etc) est faite par l'utilisation de moteurs physiques qui sont des outils approuvés et éprouvés par les professionnels du domaine. Le fait que SAMP soit intégré dans un moteur de jeu permet de pouvoir modéliser l'environnement à l'aide d'une interface graphique comme on modéliserait un niveau de jeu vidéo. En se référant à la figure 9.1, SAMP utilise le *SDK* pour accéder aux fonctions du moteur *Shine Engine* et est intégré comme un plugin à ce moteur pour être utilisable dans des jeux vidéo, mais aussi dans l'éditeur *Shine Editor*.

Malgré une interface entièrement graphique, il est nécessaire de prendre en main le langage graphique SAMP-M afin de pouvoir modéliser les comportements des agents de SAMP. Mais la possibilité d'exporter/importer des modèles de comportement d'un projet à un autre permet de différencier deux types d'utilisateurs : d'un côté des utilisateurs ayant une connaissance de la logique de programmation (qui comprendra les concepts de programmation tel que les boucles, les conditions, les tableaux, etc) et de l'autre des utilisateurs ayant une connaissance d'une logique métier (devant assembler des modèles entre eux pour modéliser le comportement du système désiré).

SAMP permet de modéliser des SMA principalement destinés au monde du jeu vidéo. Mais nous avons abordé le fait qu'il était possible d'utiliser SAMP dans des projets très éloignés de ce domaine. Le système proies-prédateurs est un classique de la littérature multi-agents et le lecteur de flux RSS n'est pas un jeu à proprement parler. Cette grande expressivité permet d'imaginer une utilisation de SAMP dans des domaines variés et de

ne pas le cloisonner au domaine des jeux vidéo qui était notre cible au départ.

9.3/ PERSPECTIVES

Les perspectives d'avenir pour SAMP sont nombreuses.

9.3.1/ OPTIMISATION DES RÈGLES DE GÉNÉRATION DE CODE

Une des premières perspectives concerne l'optimisation des règles de génération de code. Cette optimisation aurait deux impacts sur le fonctionnement de SAMP :

Le premier impact se trouverait au niveau de la compilation du code généré. En évitant aux compilateurs d'avoir à réaliser certaines optimisations en les incluant directement dans le code, il serait possible de gagner un temps précieux lors des compilations. Ces optimisations pourraient être contextuelles : Imaginons le cas de deux blocs *if* imbriqués, l'optimisation permettrait de ne générer qu'un seul bloc avec une seule condition.

Le deuxième impact serait sur l'exécution à proprement parler des SMA modélisés avec SAMP en optimisant le code généré. Tout comme pour le point précédent, il pourrait s'agir d'optimisations contextualisées : par exemple, un parcours de boucle dont la condition peut être optimisée dans un cas particulier, ou une écriture répétée en mémoire qui pourrait être évitée.

9.3.2/ FACILITÉ D'UTILISATION

Un autre point sur lequel nous avons envie de travailler est la facilité d'utilisation de SAMP. Faciliter l'utilisation de SAMP peut passer par la création de nœuds particuliers permettant de générer du code complexe qui, sans ces nœuds particuliers, nécessiterait l'utilisation de plusieurs nœuds. On a déjà évoqué un tel nœud avec le nœud *foreach* qui permet le parcours d'un tableau plus facilement qu'avec un nœud *for*. L'objectif est d'en ajouter plus afin de rendre les concepts propres au développement informatique plus accessibles et par conséquent rendre plus facile l'utilisation de SAMP.

La facilité d'utilisation de SAMP est aussi permise grâce à la possibilité d'importer/exporter des modèles d'un projet à un autre permettant à des utilisateurs débutants d'utiliser des modèles complexes modélisés par des utilisateurs confirmés. Dans sa version actuelle, SAMP permet cette importation/exportation de manière manuelle. *Shine Engine* est en évolution constante et une des évolutions majeures qui est en cours concerne le déploiement de *Shine Engine* en *SAAS*. Cette version *SAAS* est l'occasion d'offrir la possibilité de mutualiser sur un serveur en ligne (accessible par tous les utilisateurs) les modèles de comportement de SAMP. Cette mutualisation permettra à la communauté d'aider les utilisateurs qui en ont besoin en leur fournissant des modèles de comportement.

De plus, il est à l'heure actuelle possible d'intégrer de nouvelles bibliothèques de code dans SAMP. C'est de cette manière que la bibliothèque de code du *SDK* de *Shine Engine* est intégrée dans SAMP. Cependant, cette intégration requiert de fortes connaissances en informatique (ajout de macro dans le code afin que le *parser* de SAMP récupère correctement les méthodes à intégrer). Améliorer cette fonctionnalité permettra à la communauté

de proposer des bibliothèques de code prêtes à l'emploi et intégrables dans divers projets SAMP.

9.3.3/ OPTIMISER LES INTERACTIONS

Mais la facilité d'utilisation n'est pas le seul point important de notre outil et l'optimisation de l'exécution ne passe pas uniquement par l'optimisation des règles de génération. Nous avons prouvé l'intérêt de notre approche d'interactions, mais il est clairement possible de l'améliorer. Le nombre de recherches dans l'environnement effectuées par les agents a été grandement réduit mais ces recherches elles-mêmes peuvent être optimisées. Dans la version actuelle, les agents jouent tous le rôle de *Directory Facilitator (DF)* en indiquant à chaque autre agent quelles interactions il peut émettre ou recevoir.

Ce fonctionnement à un coût du fait qu'il est nécessaire de chercher dans l'environnement quels agents sont à proximité. On pourrait réduire ce coût en appliquant un concept propre aux jeux vidéo, les *chunks*. Il s'agit d'un concept consistant en des zones de l'environnement qui sont désactivées lorsqu'il n'est pas utile de mettre à jour leur état (notamment, si le joueur est trop loin de cette zone). Nous envisageons d'ajouter des agents de type *chunks* qui joueraient le rôle de *DF* et qui seraient dispersés dans l'environnement. Ainsi, chaque agent pourrait connaître les agents qui sont à proximité de lui en contactant l'agent *chunk* du *chunk* dans lequel il se trouve. Le temps de calcul pour effectuer des recherches dans l'environnement serait grandement réduit. Il serait alors possible d'appliquer des algorithmes de positionnement dans des environnements discrets afin d'obtenir un premier *tri* des agents dans l'environnement et de repasser à une recherche dans un environnement continu, mais uniquement à l'intérieur du *chunk*. Les agents actifs de SAMP exécuteraient leurs recherches uniquement auprès de ces agents *chunk*.

Un autre avantage de l'utilisation de ce concept de *chunks* serait la capacité de paralléliser les SMA de SAMP dans lesquels l'environnement serait divisé en plusieurs *chunks* et déployer ces *chunks* sur des serveurs différents. Dans le cas de jeux vidéo de type Jeu de Rôle Massivement Multi-joueurs (*MMORPG*) par exemple. Il faudrait cependant mettre en place des algorithmes de synchronisation et de propagation afin de pouvoir faire transiter des agents d'un *chunk* à l'autre.

L'intégration des *chunks* au sein de SAMP devra être transparente pour l'utilisateur. Il sera donc nécessaire de prévoir une interface simple à prendre en main pour paramétrer les *chunks*.

9.3.4/ CHANGER L'ORGANISATION

L'intégration de certains concepts agents n'est pas encore aboutie. Il a été longtemps discuté de la nécessité ou non d'intégrer des concepts tels que l'approche Agent/Groupe/-Rôle ou des types de communications particuliers comme les phéromones. Mais avant d'intégrer ce genre de concept, nous souhaitions avoir une plateforme générique permettant aux utilisateurs de modéliser ces concepts eux-mêmes. L'intégration native dans SAMP serait un véritable plus pour aider à l'accessibilité de la plateforme aux scientifiques désireux de modéliser des SMA, mais n'ayant pas les connaissances suffisantes pour le faire avec un langage informatique textuel.

L'ajout, par exemple, du concept *AGR* apporterait un véritable plus dans la gestion de SMA avec un grand nombre d'agents (comme dans les jeux de stratégie). A l'aide de ce concept, il serait possible d'améliorer l'exécution des SMA. Chaque groupe posséderait un *chef* qui centraliserait les comportements propres au groupe (déplacement, perception, etc). Ainsi, chaque agent du groupe n'aurait pas à réaliser les calculs de ces comportements ce qui réduirait grandement le nombre de calculs effectués.

Il serait possible de paramétrer et modéliser le fonctionnement interne de chaque groupe comme on paramètre et modélise des agents. On pourrait ainsi modéliser les conditions d'accès au groupe, le comportement des agents et du groupe en cas de mort de leur *chef* (dissolution du groupe, élection d'un nouveau *chef*). Imaginons un groupe qui perdrait un grand nombre de ses agents sous les coups d'un adversaire trop puissant. Il serait envisageable de modéliser la dissolution du groupe lorsque le nombre d'agent en son sein passerait sous un certain seuil. Ainsi, les agents seraient mis en déroute et fuiraient le champ de bataille.

L'ajout de ce concept *AGR* aurait un impact tant d'un point de vue de l'optimisation de l'exécution que de la centralisation des comportements et leur réalisme.

Un système tel que SAMP sera toujours en évolution et de nouvelles perspectives seront mises en avant par les utilisateurs. Mais les perspectives citées dans cette section nous semblent être de bons objectifs à atteindre afin d'améliorer SAMP et le rendre plus accessible et plus performant.

9.4/ UN PETIT MOT POUR LA FIN

Je terminerai par une analyse plus personnelle. Ces trois années m'ont permis d'acquérir des connaissances dans un domaine extrêmement vaste et complexe. Les systèmes multi-agents apportent des possibilités et des solutions à de nombreux problèmes. Ces nouvelles connaissances et ce travail de recherche m'ont fait prendre conscience qu'il y a une multitude de possibilités pour répondre à un problème, et ce bien avant l'implémentation, mais déjà lors de la réflexion sur les architectures, les technologies et les concepts qui seront utilisés.

J'ai déjà remercié toutes les personnes qui ont pris part à l'aboutissement de cette thèse, mais je tenais à vous réaffirmer toute la sympathie que j'ai pour vous.

BIBLIOGRAPHIE

[1] Extension gis version 1.0, diffusée le 16.07.2008. http ://romainmejean.fr/manuel_-netlogo/gis.html.

[2] Gama, modeling made easy. http ://gama-platform.org/. 2016.

[3] Patrice langlois, baptiste blanpain et eric daudé, « magéo, une plateforme de modélisation et de simulation multi-agent pour les sciences humaines », cybergeo : European journal of geography [en ligne], systèmes, modélisation, géostatistiques, document 741, mis en ligne le 02 octobre 2015, consulté le 23 décembre 2016. url : http ://cybergeo.revues.org/27236 ; doi : 10.4000/cybergeo.27236.

[4] Fipa request interaction protocol specification. *http ://www.fipa.org/specs/fipa00026/*, 2002.

[5] Z. Afoutni. *Un modèle multi-agents pour la représentation de l'action située basé sur l'affordance et la stigmergie*. PhD thesis, Université de la Réunion, 2015.

[6] E. Amouroux, T.-Q. Chu, A. Boucher, and A. Drogoul. Gama : an environment for implementing and running spatially explicit multi-agent simulations. In *Pacific Rim International Conference on Multi-Agents*, pages 359–371. Springer, 2007.

[7] F. Arrignon. *HOVER-WINTER : un modèle multi-agent pour simuler la dynamique hivernale d'un insecte auxiliaire des cultures (Episyrphus balteatus, Diptera : Syrphidae) dans un paysage hétérogène*. PhD thesis, 2006.

[8] J. L. Austin. *How to do things with words*. Oxford university press, 1975.

[9] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 95–105. IEEE, 2001.

[10] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5) :584–600, 2004.

[11] R. Bajcsy. Active perception. *Proceedings of the IEEE*, 76(8) :966–1005, 1988.

[12] J. C. Ballantyne. Real-time scheduler, Oct. 18 2005. US Patent 6,957,432.

[13] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6) :139–143, 2010.

[14] S. Bernardini, K. Porayska-Pomsta, and T. J. Smith. Echoes : An intelligent serious game for fostering social communication in children with autism. *Information Sciences*, 264 :41–60, 2014.

[15] V. Berry. Jouer pour apprendre : est-ce bien sérieux ? réflexions théoriques sur les relations entre jeu (vidéo) et apprentissage. *Canadian Journal of Learning and Technology/La revue canadienne de l'apprentissage et de la technologie*, 37(2), 2011.

[16] T. Bouron and A. Collinot. Sam : a model to design computational social agents. In *Proc. 10th European Conference on Artificial Intelligence, ECAI*, volume 92, pages 239–243, 1992.

[17] T. Bouron, J. Ferber, and F. Samuel. Mages : A multi-agent testbed for heterogeneous agents. *Decentralized Artificial Intelligence*, 2 :195–214, 1991.

[18] F. Bousquet, I. Bakam, H. Proton, and C. Le Page. Cormas : common-pool resources and multi-agent systems. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 826–837. Springer, 1998.

[19] Box2D. A 2d physics engine for games. http://box2d.org/, 2018.

[20] P. Breiter and M. Sadek. A rational agent as a kernel of a co-operative dialogue system. In *Proceedings EC AT 96 ATAL Workshop*.

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

[22] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations : an algorithm and a tool. In *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on*, pages 85–94. IEEE, 2006.

[23] G. Caire, W. Coulier, F. Garijo, J. Gomez, J. Pavón, F. Leal, P. Chainho, P. Kearney, J. Stark, R. Evans, et al. Agent oriented analysis using message/uml. In *International Workshop on Agent-Oriented Software Engineering*, pages 119–135. Springer, 2001.

[24] S. Chandhoke. Hardware assisted real-time scheduler using memory monitoring, Jan. 27 2015. US Patent 8,943,505.

[25] P. Dagnelie. *Principes d'expérimentation (deuxième édition)*. Presses agronomiques de Gembloux, 2012.

[26] Y. Demazeau. From interactions to collective behaviour in agent-based systems. In *In : Proceedings of the 1st. European Conference on Cognitive Science. Saint-Malo*. Citeseer, 1995.

[27] A. Drogoul. When ants play chess (or can strategies emerge from tactical behaviours ?). In *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 11–27. Springer, 1993.

[28] A. Drogoul. *When ants play chess (Or can strategies emerge from tactical behaviours ?)*, pages 11–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[29] E. H. Durfee and T. A. Montgomery. Mice : A flexible testbed for intelligent coordination experiments. *Ann Arbor*, 1001 :48103, 1989.

[30] E. H. Durfee and T. A. Montgomery. A hierarchical protocol for coordinating mulitagent behaviors. In *AAAI*, pages 86–93, 1990.

[31] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2) :114–131, 2003.

[32] J. Ferber. *Les systèmes multi-agents : vers une intelligence collective*. Informatique, Intelligence Artificielle. Inter-éditions, 1995.

[33] J. Ferber. Les systèmes multi-agents : un aperçu général. *Techniques et sciences informatiques*, 16(8), 1997.

[34] R. E. Fikes and N. J. Nilsson. Strips : A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4) :189–208, 1971.

[35] J. B. Filippi. *Une architecture logicielle pour la multi-modélisation et la simulation à évènements discrets de systèmes naturels complexes*. PhD thesis, Université de Corse ; Université Pascal Paoli, 2003.

[36] T. Finin, R. Fritzson, D. McKay, and R. McEntire. Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM, 1994.

[37] F. for Intelligent Physical Agent. Foundation for intelligent physical agent - specifications. *http ://www.fipa.org/specifications/*, 1997.

[38] A. U. Frank, S. Bittner, and M. Raubal. Spatial and cognitive simulation with multi-agent systems. In *Proceedings of the International Conference on Spatial Information Theory : Foundations of Geographic Information Science*, pages 124–139. Springer-Verlag, 2001.

[39] B. P. Gerkey and M. J. Matarić. Murdoch : Publish/subscribe task allocation for heterogeneous agents. In *Proceedings of the fourth international conference on Autonomous agents*, pages 203–204. ACM, 2000.

[40] J. J. Gibson. *The ecological approach to visual perception : classic edition*. Psychology Press, 2014.

[41] A. Gouaïch, F. Michel, and Y. Guiraud. Mic* : a deployment environment for autonomous agents. In *International Workshop on Environments for Multi-Agent Systems*, pages 109–126. Springer, 2004.

[42] A. Grignard, P. Taillandier, B. Gaudou, D. A. Vo, N. Q. Huynh, and A. Drogoul. Gama 1.6 : Advancing the art of complex agent-based modeling and simulation. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 117–131. Springer, 2013.

[43] C. Grossard and O. Grynszpan. Entraînement des compétences assistées par les technologies numériques dans l'autisme : une revue. *Enfance*, (1) :67–85, 2015.

[44] O. M. Group. Metaobject facility, 2018.

[45] Y. HAFRI and N. M. NAJID. Utilisation de l'approche multi-agents pour le pilotage en temps réel des systèmes de production. In *3e Conférence Francophone de Modélisation et Simulation : Conception, Analyse et Gestion des Systèmes Industriels, MOSIM*, volume 1, page 25, 2001.

[46] E. Hermellin, F. Michel, and J. Ferber. Systèmes multi-agents et gpgpu : état des lieux et directions pour l'avenir. In *JFSMA : Journées Francophones sur les Systèmes Multi-Agents*, pages 97–106, 2014.

[47] N. Hocine, A. Gouaïch, I. Di Loreto, and L. Abrouk. Techniques d'adaptation dans les jeux ludiques et sérieux. *Revue des Sciences et Technologies de l'Information-Série RIA : Revue d'Intelligence Artificielle*, 25(2) :253–280, 2011.

[48] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl : A model transformation tool. *Science of computer programming*, 72(1-2) :31–39, 2008.

[49] V. Julian and V. Botti. Developing real-time multi-agent systems. *Integrated Computer-Aided Engineering*, 11(2) :135–149, 2004.

[50] I. Kosonen and A. Bargiela. Simulation based traffic information system. In *Seventh World Congress on Intelligent Transport Systems*, pages 6–9, 2000.

[51] Y. Kubera, P. Mathieu, and S. Picault. Everything can be agent ! In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems : volume 1-Volume 1*, pages 1547–1548. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[52] Y. Kubera, P. Mathieu, and S. Picault. Ioda : an interaction-oriented approach for multi-agent based simulations. *Autonomous Agents and Multi-Agent Systems*, 23(3) :303–343, 2011.

[53] Y. Labrou and T. Finin. A proposal for a new kqml specification. Technical report, Technical Report Technical Report TR-CS-97-03, University of Maryland Baltimore County, 1997.

[54] Y. Labrou and T. Finin. Semantics for an agent communication language. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 209–214. Springer, 1997.

[55] P. Langlois, B. Blanpain, and E. Daudé. Magéo, une plateforme de simulation multi-agents pour tous. In *SimTools 2013*, 2013.

[56] P. Mahot and A. Nédélec. Communication entre agents informatiques dans un environnement virtuel. 2005.

[57] P. Mathieu, S. Picault, and Y. Secq. Design patterns for environments in multi-agent simulations. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 678–686. Springer, 2015.

[58] D. V. McGehee, E. N. Mazzae, and G. S. Baldwin. Driver reaction time in crash avoidance research : Validation of a driving simulator study on a test track. In *Proceedings of the human factors and ergonomics society annual meeting*, volume 44 of *20*, pages 320–323. SAGE Publications Sage CA : Los Angeles, CA, 2000.

[59] D. McKay and R. McEntire. Kqml-a language and protocol for knowledge and information exchange. Technical report, Technical Report CS-94-02, Computer Science Department, University of Maryland and Valley Forge Engineering Center, Unisys Corporation, 1994.

[60] R. Neches, R. E. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout. Enabling technology for knowledge sharing. *AI magazine*, 12(3) :36, 1991.

[61] A. Nicolle. Le continu, le discontinu et le discret en informatique. *Espaces Temps*, 82(1) :97–109, 2003.

[62] P. D. O'Brien and R. C. Nicol. Fipa-towards a standard for software agents. *BT Technology Journal*, 16(3) :51–59, 1998.

[63] J. Oldevik, T. Neple, R. Grønmo, J. Aagedal, and A.-J. Berre. Toward standardised model to text transformations. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 239–253. Springer, 2005.

[64] H. V. D. Parunak. " go to the ant" : Engineering principles from natural multi-agent systems. *Annals of Operations Research*, 75 :69–101, 1997.

[65] B. Physics. Real-time physics simulation. http://bulletphysics.org/wordpress/, 2018.

[66] J. Real and A. Crespo. Mode change protocols for real-time systems : A survey and a new proposal. *Real-time systems*, 26(2) :161–197, 2004.

[67] P. Richmond and D. Romano. Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu. In *Proceedings International Workshop on Supervisualisation*, 2008.

[68] J.-C. Routier, P. Mathieu, Y. Secq, et al. Dynamic skill learning : A support to agent evolution. In *Proceedings of the AISB01 Symposium on Adaptive Agents and Multi-Agent Systems*, pages 25–32, 2001.

[69] S. Russell, P. Norvig, and A. Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25 :27, 1995.

[70] S. J. Russell and D. Subramanian. Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, 2 :575–609, 1995.

[71] T. Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *AAAI*, volume 93, pages 256–262, 1993.

[72] T. Sandholm, V. R. Lesser, et al. Issues in automated negotiation and electronic commerce : Extending the contract net framework. In *ICMAS*, volume 95, pages 12–14, 1995.

[73] S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot. Using models of partial knowledge to test model transformations. In *International Conference on Theory and Practice of Model Transformations*, pages 24–39. Springer, 2012.

[74] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory : A historical perspective. *Real-time systems*, 28(2-3) :101–155, 2004.

[75] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 134–139. ACM, 1999.

[76] Y. Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1) :51–92, 1993.

[77] M. P. Singh. Agent communication languages : Rethinking the principles. *Computer*, 31(12) :40–47, 1998.

[78] R. G. Smith. The contract net protocol : High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, (12) :1104–1113, 1980.

[79] L. Steels. The origins of ontologies and communication conventions in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 1(2) :169–194, 1998.

[80] P. Taillandier. Gamagram : Modélisation graphique sous gama. In *Masyco 2013*, 2013.

[81] P. Taillandier, D.-A. Vo, E. Amouroux, and A. Drogoul. Gama : a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 242–258. Springer, 2010.

[82] S. Tisue and U. Wilensky. Netlogo : A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA, 2004.

[83] L. Valente, A. Conci, and B. Feijó. Real time game loop models for single-player computer games. In *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, volume 89, page 99, 2005.

[84] J. Van den Berg, M. Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 1928–1935. IEEE, 2008.

[85] B. Virole. *Du bon usage des jeux vidéo et autres aventures virtuelles*. Hachette littératures, 2003.

[86] D. Weyns, E. Steegmans, and T. Holvoet. Towards active perception in situated multi-agent systems. *Applied Artificial Intelligence*, 18(9-10) :867–883, 2004.

[87] U. Wilensky. Netlogo. *http ://ccl.northwestern.edu/netlogo/*, 1999.

[88] M. Wooldridge and N. R. Jennings. Intelligent agents : Theory and practice. *The knowledge engineering review*, 10(2) :115–152, 1995.

[89] P. Xuan, V. Lesser, and S. Zilberstein. Communication decisions in multi-agent cooperation : Model and experiments. In *Proceedings of the fifth international conference on Autonomous agents*, pages 616–623. ACM, 2001.

[90] B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of modeling and simulation*. Academic press, 2000.

[91] H. Zgaya. *Conception et optimisation distribuée d'un système d'information d'aide à la mobilité urbaine : Une approche multi-agent pour la recherche et la composition des services liés au transport*. PhD thesis, Ecole Centrale de Lille, 2007.

TABLE DES FIGURES

LISTE DES TABLES

Document réalisé avec LaTeX et :
le style LaTeX pour Thèse de Doctorat créé par S. Galland — http://www.multiagent.fr/ThesisStyle
la collection de paquets `tex-upmethodology` — http://www.arakhne.org/tex-upmethodology/