Optimiser l'évaluation du contexte dans une simulation multi-agent

Flavien Balbo, Mahdi Zargayouna, Fabien Badeig

Optimiser l'évaluation du contexte dans une simulation multi-agent

Flavien Balbo[a] flavien.balbo@mines-stetienne.fr
Mahdi Zargayouna[b] hamza-mahdi.zargayouna@ifsttar.fr
Fabien Badeig[a] fabien.badeig@mines-stetienne.fr

[a] Institut Henri Fayol, ENS Mines Saint-Etienne, Saint-Etienne, France.

[b] Université Paris-Est, IFSTTAR, GRETTIA, Champs sur Marne, France.

Abstract

L'exécution d'une simulation multi-agent (MABS) nécessite un ordonnanceur afin de synchroniser l'exécution des agents et simuler la simultanéité de leurs comportements. Dans la majorité des frameworks *MABS, l'ordonnanceur active les agents tour à tour afin qu'ils exécutent une action décidée selon leur contexte. L'agent activé utilise toutes les informations sur lui-même, les autres agents ou objets de son environnement qui lui sont accessibles. Face à ce volume de données, une difficulté est de déterminer efficacement les combinaisons qui font sens pour l'agent parce qu'elles caractérisent un contexte pertinent pour lui. Dans la majorité des MABS, la détermination de ces contextes est enfouie dans le code des agents et par conséquent il n'existe pas d'algorithme permettant de diminuer leur temps de calcul. Nous proposons de modéliser ces sous-ensembles d'informations identifiant un contexte par ce que nous appelons un filtre ainsi qu'un algorithme pour que chaque agent puisse efficacement déterminer les filtres qui le concernent. Nous comparons notre algorithme à une sélection séquentielle des filtres et discutons les premiers résultats.*

Mots-clés : *simulation multi-agent, contexte, modèle d'agents*

1 Introduction

Un des objectifs principaux d'une simulation est la reproduction contrôlée de systèmes complexes. La simulation de la simultanéité qui implique que plusieurs agents puissent agir au même pas de temps fait partie du contrôle à assurer. Par conséquent, similairement à la gestion multitâche d'un système d'exploitation, l'exécution d'une simulation multi-agent nécessite un processus d'ordonnancement afin de synchroniser l'exécution des agents et simuler la simultanéité par un contrôle de l'exécution d'un pas de temps et de l'évolution du temps. La majorité des plateformes de simulation suit un modèle collaboratif d'ordonnancement où l'activation des agents est contrôlée par l'ordonnanceur et leur interruption par les agents eux mêmes. Après avoir été activé, l'agent détermine son contexte et décide de l'action à réaliser.

La participation des agents au processus d'ordonnancement a pour avantage de clairement identifier l'étape où les contextes sont calculables : à l'activation de l'agent, l'état de la simulation lui est partiellement connu et ne sera éventuellement modifié que par son action. La contrepartie est un potentiel coût important en termes de temps de calcul car l'agent doit simultanément prendre en compte toutes les informations qui lui sont accessibles, c'est à dire les informations issues de sa perception de l'environnement (les autres agents et objets) ainsi que de son propre état [5]. La difficulté est d'identifier les combinaisons de ces informations qui font sens pour l'agent car elles caractérisent un contexte. La définition de ces contextes appartient à la connaissance de l'agent puisqu'elle conditionne son comportement. Le problème est que cette connaissance est souvent enfouie dans le code des agents et la détermination du contexte courant est réalisée en même temps que la recherche de l'action à exécuter. Il devient alors difficile de personnaliser pour chaque agent la définition de son contexte sans avoir à modifier son implémentation.

Pour diminuer ce coût, le concepteur utilise classiquement des tests imbriqués et/ou des automates de comportement. De notre point de vue, l'utilisation de tests imbriqués augmente la complexité de conception des agents et la difficulté de personnaliser ou modifier dynamiquement sa prise en compte du contexte. L'utilisation d'un automate de comportement donne une part très importante à une information agrégée

qu'est l'état de l'agent et conditionne la prise en compte d'autres informations dans la détermination du contexte d'un agent.

Nous proposons de modéliser les contextes comme des *filtres* afin d'expliciter cette partie des connaissances des agents. Chaque agent a ses propres filtres et exécute un algorithme que nous décrivons dans cet article afin de trouver efficacement tous les filtres possibles en fonction des informations accessibles.

La suite de cet article est organisée comme suit : dans la section 2, nous discutons les enjeux concernant la détermination du contexte ; dans la section 3, nous présentons un exemple illustratif de la complexité pour modéliser le contexte ; dans la section 4, nous définissons formellement notre proposition ; dans la section 5, nous détaillons notre algorithme de détermination du contexte ; dans la section 6 nous présentons nos expérimentations et résultats ; enfin, nous concluons avec une discussion de notre proposition et les perspectives que nous envisageons.

2 État de l'art

La détermination du contexte est conditionnée par l'information à laquelle l'agent activé accède. Dans cette section, nous présentons les approches pour la détermination du contexte.

La première approche, la plus courante, est centrée agent. L'ordonnanceur active les agents en appelant une méthode par défaut ([2, 3, 11, 15]) ou par un message de contrôle ([6, 14, 16]) et chacun détermine son contexte. Par exemple, dans [2], les objets se trouvant dans le champ de perception de l'agent lui sont fournis par un évènement perception. Ainsi, le lien entre le comportement des agents et l'ordonnanceur est minimal et standardisé. Nous pouvons également mettre dans cette approche les plateformes multi-agents à la NetLogo comme Madkit [8] ou StarLogo [1]. Dans ce cas l'agent est activé selon l'état de son automate et donc dans un contexte prédéfini par l'agent lors de l'activation précédente.

La seconde approche est centrée ordonnanceur. Celui-ci détermine pour chaque agent son contexte et l'action associée. A notre connaissance, les plateformes JEDI [10], Repast Simphony [4] et nos précédents travaux [1] sont les seules propositions où le choix de l'action réalisée par l'agent est fait par l'ordonnanceur. Dans le framework JEDI [10], le choix de l'action est issu de l'analyse d'une matrice d'interaction où chaque cellule représente une interaction conditionnée entre deux agents. Les conditions portent sur l'état de la simulation, c'est à dire le contexte. Par exemple, une interaction est possible entre deux agents selon leur proximité. Une action est associée à chacun de ces contextes et elle sera exécutée par l'agent activé. Cette approche utilise une réification d'une interaction et son traitement peut alors être externalisé [12]. L'avantage est que la gestion des interactions peut alors être optimisée ou simplement modifiée puisque cette partie de l'activité d'un agent n'est pas noyée dans son code. Le choix de la structure de matrice limite à deux le nombre de type d'agents concernés par une interaction. Le fonctionnement courant de Repast Simphony utilise un ordonnanceur sans information. Cependant, il permet également l'utilisation de *watchers* qui notifient un agent de la modification de l'état d'un autre agent et déclenche une action associée. Le concepteur définit quel agent est surveillé et les conditions à valider pour déclencher l'action associée. Ce mécanisme est pénalisé par la faible expressivité du langage pour exprimer les conditions du contexte. Dans nos précédents travaux, l'environnement est utilisé comme ordonnanceur et active les agents selon des filtres qui modélise leur contexte. Nous avons proposé qu'un contexte soit modélisé par des conditions sur les informations fournies par l'environnement sur les composants de la simulation. Nous reprenons notre modélisation par des filtres et proposons un nouvel algorithme de détermination du contexte exécuté par chaque agent et non par l'environnement.

3 Exemple illustratif

Pour illustrer notre proposition, nous prenons pour exemple le comportement d'un conducteur qui entre dans un rond point. Notre objectif est de souligner la complexité de la modélisation du contexte. La figure 1 représente un rond-point avec des agents véhicules, piétons et des objets comme les éléments de signalisation.

Un contexte est une combinaison des informations perceptibles qui est pertinente pour l'agent. Par exemple, un agent piéton p_1 est perçu par les agents véhicules v_1 et v_2 (figure 1) mais le contexte résultat est différent. Ainsi pour v_1 le contexte peut être *" je vais entrer dans le rond-point mais ma vitesse est excessive et il y*

1. http ://education.mit.edu/StarLogo/

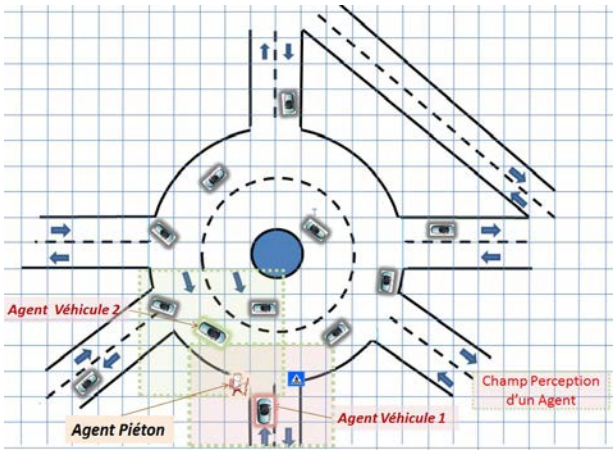

FIGURE 1 – Exemple de simulation de rond point

a un piéton p_1 qui traverse devant moi" alors que pour v_2, le contexte peut être *"je suis sur la voie extérieure du rond-point et je vais croiser une voie entrante qui est bloquée par le piéton p_1"*. Les informations partagées peuvent donc être les mêmes mais c'est leur combinaison qui fait sens pour chaque agent. Le problème est que ces combinaisons relèvent de l'expertise, qu'elles sont multiples et que leur calcul est très coûteux car réalisé pour chaque agent à chaque pas de temps. Il faut donc proposer une solution qui soit à la fois efficace mais qui permette également d'exprimer toute la complexité des contextes possibles. Une fois qu'un agent a déterminé les contextes pertinents, il doit alors décider de l'action à exécuter. Notre proposition concerne l'identification efficace de ces contextes parmi l'ensemble des contextes potentiels selon les informations perceptibles par les agents. Le modèle d'agent pour utiliser cette connaissance n'est pas contraint. Par exemple, v_1 peut être un agent BDI qui planifiera ses actions et v_2 un agent réactif qui freinera.

4 Définition du modèle

La détermination du contexte suppose de pouvoir donner à l'agent activé les informations sur les composants du SMA qui lui sont accessibles. Les conditions d'accessibilité sont à définir pour chaque simulation et nous considérons ici que chaque agent a un champ de perception dans lequel il perçoit tous les composants du SMA. Dans cette section, nous proposons un modèle de données afin de modéliser un contexte à partir des informations perçues.

Le premier composant du modèle est appelé entité et constitue une méta-information sur un composant du SMA.

Definition 1 (Entité) *Une* entité $\omega \in \Omega$ *est un couple* $\langle r_\omega, d_\omega \rangle$ *avec :*
- r_ω *la référence d'un composant réel du système multi-agent (agent ou objet).*
- d_ω *la description de ce composant enregistrée dans l'environnement. Elle est définie par un ensemble de couples* $\langle propriété, valeur \rangle$.

r_ω permet à l'environnement d'accéder au composant réel (pour l'activer par exemple si c'est un agent) ; d_ω est utilisée pour identifier le contexte des agents. Une entité fait le lien entre le SMA et le monde modélisé pour la détermination des contextes. Chaque composant du SMA devant être pris en compte par l'environnement afin d'identifier un contexte doit être associé à une entité. Pour chacun d'eux, il faut définir les propriétés qui seront prises en compte. Dans la suite, sauf indication contraire, nous assimilerons l'entité à sa description. Une propriété donne une information sur une entité avec $\mathbb{D} = \{d_1, ..., d_m\}$ l'ensemble des domaines de description des propriétés.

Definition 2 (Propriété) *Une propriété $p_i \in \mathcal{P}$ est une fonction dont le domaine de description $d_j \in \mathbb{D}$ peut être quantitatif, qualitatif, ou un ensemble fini de données :* $p_i : \Omega \rightarrow d_j$

Ces propriétés peuvent être regroupées afin de caractériser des sous-ensembles d'entités.

Definition 3 (PDescription) *Une PDescription est un sous-ensemble de $\mathcal{P}$ et notons P_e la PDescription de l'entité e.*

L'extension d'une *PDescription* pour une entité est appelée *Catégorie*.

Definition 4 (*Catégorie*) *Une Catégorie C_x est un ensemble d'entités avec la même PDescription et qui sont sémantiquement similaires :* $C_x = \langle label, \{\omega \in \Omega | P_{\omega_i} = P_{\omega_j} \forall \omega_i, \omega_j \in C_x\} \rangle$ *avec label le nom de la Catégorie.*

Dans notre exemple, les agents véhicules (VA), agents piétons (PA) et objets de signalisation (TS) sont des exemples de Catégorie. Enfin, la liste des données accessibles (entités) fournies par l'environnement qui constitue le point d'entrée de notre algorithme est appelée *CategorieAccessible* et ne contient que des Catégories non vides.

Par exemple, la description d'un agent véhicule pourrait être (nous notons $\Omega_\mathcal{A} \subset \Omega$ l'ensemble des agents) :

– *vitesse* : $\Omega_{\mathcal{A}} \rightarrow \mathbb{R}$ donne la vitesse ;
– *topologie* : $\Omega_{\mathcal{A}} \rightarrow$ { *rues*, *voies du rond point*} donne la position selon la topologie du réseau.
– *position* : $\Omega_{\mathcal{A}} \rightarrow \mathbb{N}$: la distance depuis l'entrée du rond point ou une valeur relative pour une voie du rond point ;
– *direction* : $\Omega_{\mathcal{A}} \rightarrow$ { *vers rond-point, depuis rond-point*} donne la direction relative depuis le rond-point ;
– *clignotant* : $\Omega_{\mathcal{A}} \rightarrow \{gauche, droit, éteint\}$: donne l'état des clignotants.
– ...

Nous proposons de modéliser un contexte comme un filtre dont la partie condition est exprimée sur les descriptions perçues par l'agent.

Definition 5 (Filtre) *Un filtre $F_j \in \mathcal{F}$ est un n-uplet $F_j = \langle f_a, f_C, n_f \rangle$ avec :*
– *$f_a : \Omega_{\mathcal{A}} \rightarrow \{V, F\}$ assertion qui exprime les contraintes sur l'agent possesseur du filtre ;*
– *$f_C : 2^{\Omega} \rightarrow \{V, F\}$ ensemble d'assertions exprimant des contraintes sur les autres composants complétant le contexte ;*
– *n_f : nom du filtre.*

Un filtre identifie par unification les descriptions d'un agent et du contexte (un sous ensemble de descriptions) qui apparient ses assertions. Un filtre est valide pour tout n-uplet $\langle agent \in \Omega_{\mathcal{A}}, context \subset \Omega \rangle$ tel que $f_a(agent) \wedge f_C(context)$ est vrai. Quand un filtre est valide, le contexte associé $\langle context, n_f \rangle$ est valide pour l'agent a. Un contexte étant formalisé comme des contraintes sur les descriptions des composants du SMA, les informations *context* et n_f sont complémentaires pour le caractériser. En effet, le même ensemble de descriptions peut valider d'autres filtres et un même filtre peut être validé par plusieurs ensembles de descriptions.

Prenons par exemple le filtre $\langle f_a, f_C, warning \rangle$ qui identifie un contexte de danger selon le mouvement de véhicules. Chaque filtre appartient à un agent et est par conséquent construit selon son point de vue. Pour le filtre *warning*, l'agent véhicule (propriétaire) est sur la voie centrale du rond point et un véhicule plus lent, devant lui sur l'autre voie a son clignotant gauche allumé. Le déclenchement de ce filtre dépend de : i) la position de l'agent (assertion f_a) ; ii) la perception d'un autre agent véhicule avec les valeurs des propriétés respectant les contraintes f_C. Le filtre *warning* a la définition suivante :
– $a \in \Omega_{\mathcal{A}}$: f_a : $[vitesse(a) = ?s_a] \wedge [topologie(a) = voieCentrale] \wedge [position(a) = ?l_a]$
– $b \in \Omega_{\mathcal{A}}$: $f_C(b)$: $[vitesse(b) < ?s_a] \wedge [position(b) < ?l_a] \wedge [clignotant(b) = gauche] \wedge [topologie(b) = voieExterne]$

Le symbole " ?" devant une expression identifie une variable et le symbole "=" est l'opérateur de comparaison. Dans ce filtre, un agent b est dans le champ de perception et est donc proche de l'agent a. C'est donc l'environnement, notre ordonnanceur, qui a pré-filtré les informations disponibles dans la simulation selon le critère de la proximité.

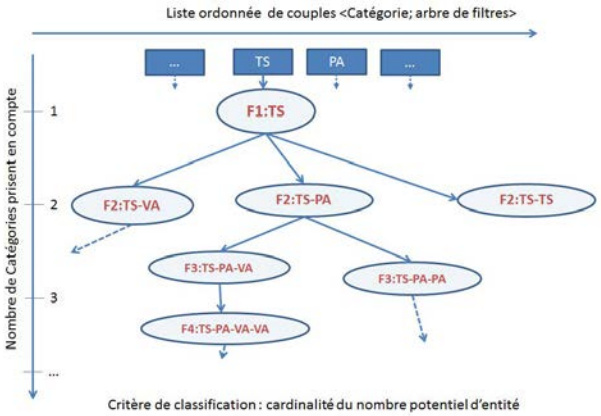

FIGURE 2 – Liste ordonnée d'arbres de filtres

Les filtres des agents sont enregistrés dans une liste ordonnée de couples que nous appelons *CategoriePertinente*. Le premier membre d'un couple est le label d'une Catégorie, que nous appelons référence, et le second membre est un arbre ordonné de filtres (présenté ci-dessous). Par exemple, la figure 2 décrit le couple (TS, arbre) pour un agent véhicule. Chaque agent a sa propre liste d'arbres de filtres et exécute l'algorithme décrit dans la section 5 pour le parcourir et identifier les filtres qui s'apparient sur les entités perceptibles.

L'évaluation d'un filtre est conditionnée par l'existence d'entités pour chacune des Catégories qu'il doit tester. La Catégorie la moins représentée conditionne donc plus l'évaluation que la seconde, qui conditionne plus que la troisième, etc. Par conséquent, nous organisons chaque arbre de filtres selon cette relation d'ordre. A une Catégorie correspondent les filtres qui la teste (assertion f_C) et qui ne testent pas une Catégorie de cardinalité inférieure. Par exemple, dans la figure 2, l'arbre de la Catégorie TS contient les filtres testant la Catégorie TS seule ou avec les Catégories *PA* et *VA*. Depuis la Catégorie *PA* (second élément de la liste), nous obtenons les filtres où la Catégorie *PA* est testée seule ou avec la Catégorie *VA*. Nous ordonnons enfin la liste *CategoriePertinente* selon le nombre potentiel d'entités qu'elle peut contenir.

Nous assumons que le nombre d'entités par Ca-

tégorie est donné par le concepteur de la simulation ou qu'il est au moins capable de les ordonner relativement. Dans l'exemple du rond-point, le concepteur donne l'ordre suivant $\cdots < |TS| < |PA| < |VA| < \ldots$ si la simulation concerne l'heure de pointe et un trafic important ou l'ordre $\cdots < |PA| < |VA| < |TS| < \ldots$ si la simulation concerne un faible trafic de nuit. Il s'agit donc de considérer un environnement ouvert dans lequel on peut relativiser le nombre d'entités par Catégorie. L'ordre est donc potentiellement faux ponctuellement mais vrai sur la durée de la simulation.

Nous utilisons une convention de nommage des filtres qui indique le niveau de profondeur du filtre dans l'arbre et respecte l'ordre des cardinalités. Par exemple, figure 2, $F2 : TS - VA$ indique qu'il s'agit d'un filtre de profondeur 2 et donc concernant deux Catégories : *TS* et *VA*. Le respect de l'ordre des cardinalités permet section 5 de chercher efficacement les filtres dont l'évaluation est possible et il assure l'unicité des filtres. Par exemple, le filtre $F2 : TS - VA$ n'appartient pas à l'arbre de la Catégories *VA*. Néanmoins, à des fins de clarté, nous donnons un nom explicite au filtre lorsque la position du filtre dans l'arbre n'est pas le sujet comme ce fut le cas pour le filtre *warning*.

Un nœud contient les filtres pour lesquels f_C teste le même ensemble de Catégories. Pour distinguer ces filtres, nous ajoutons une lettre au nom du filtre. Par exemple, le nœud $F4 : PA - TS - VA - VA$ (figure 2) contient tous les filtres pour lesquels f_C est à valider avec les descriptions d'un agent piéton, d'un élément de signalisation et deux agents véhicules (en plus de l'agent véhicule propriétaire de l'arbre de filtres). Le filtre *warning* appartient au nœud $F1 : VA$ puisque f_C concerne un agent véhicule.

Un arc représente la relation d'inclusion entre deux nœuds : le nœud le plus profond (le fils) contient des filtres dont l'évaluation suppose une Catégorie de plus à tester que le nœud le moins profond (le père). Par exemple, les enfants de $F1 : TS$ sont $F1 : TS - VA$, $F1 : TS - PA$ et $F1 : TS - TS$ avec respectivement l'ajout des Catégories *VA, PA* et *TS*.

Pour une profondeur donnée, nous ordonnons les nœuds selon l'ordre décroissant des cardinalités. En effet, les enfants d'un nœud sont explorés si la Catégorie supplémentaire appartient à la liste des Catégories accessibles (section 5). Par conséquent, traiter en priorité les enfants ayant potentiellement le plus de descriptions accroît la possibilité d'obtenir un contexte valide et d'arrêter la recherche. Par exemple, il y a potentiellement plus de véhicules que de piétons qui sont perçus par un agent véhicule. Par conséquent, dans la figure 2, les filtres appartenant au nœud $F3 : TS - PA - VA$ sont testés avant ceux appartenant au nœud $F3 : TS - PA - PA$ si la Catégorie *VA* appartient à l'ensemble des Catégories perceptibles. Si l'objectif est de retrouver tous les contextes possibles d'un agent, alors l'ordre des noeuds n'a pas d'importance.

Enfin, si un enfant n'a pas de parent, c'est à dire qu'il n'existe pas de filtres concernant les Catégories de ses parents, alors le nœud parent est créé vide.

A partir de cette structure de filtres et des descriptions, nous concevons un algorithme permettant d'identifier efficacement les filtres possibles selon les entités perceptible depuis l'environnement.

5 Algorithme de calcul du contexte

L'objectif de l'algorithme est de ne tester que les filtres pour lesquelles il existe des descriptions accessibles à l'agent. Si une Catégorie est représentée, les filtres contenus à la racine de l'arbre associé sont à tester, puis les filtres contenus dans chacun de ses successeurs si la Catégorie ajoutée existe dans la liste fournie par l'environnement. Nous dissocions la condition d'existence des Catégories à tester des conditions exprimées dans les filtres. Ainsi, un nœud peut être examiné selon la condition d'existence et n'avoir aucun filtre validé selon les conditions exprimées dans les filtres alors qu'un enfant peut avoir des filtres validés selon les conditions d'existence et des filtres. Par exemple, un filtre appartenant au nœud $F3 : TS - PA - PA$ peut être valide alors qu'aucun filtre dans le nœud $F3 : TS - PA$ ne l'est (figure 3). Notre structure de données est construite afin d'exploiter la nature des données perçues (condition d'existence) et non leur état (condition des filtres). L'avantage est d'être indépendant de la dynamique de l'environnement et d'éviter de coûteuses mises à jour comme c'est le cas d'algorithmes comme Rete [9] qui traitent simultanément les deux contraintes.

Dans l'algorithme 1 d'ordonnancement, un nombre de pas de temps est fixé (T) et nous activons tous les agents à chaque pas de temps en leur fournissant une liste *CategorieAccessible*. Cette liste est construite par l'environnement en

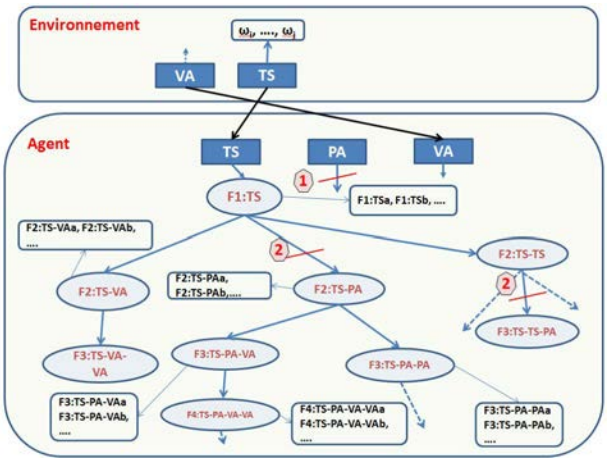

FIGURE 3 – Aperçu global du fonctionnement de l'algorithme

sélectionnant les entités accessibles et qui sont pertinentes pour l'agent (1-(4)) car prises en compte dans ses filtres. Cette liste n'est pas triée car notre algorithme a pour objectif de fournir tous les contextes possibles et il est donc nécessaire de parcourir tous les arbres possibles. Ainsi, nous ne contraignons pas le processus de décision qui sur la base des contextes possibles aboutit à la sélection d'une action. Nous notons $\mathcal{A}$ l'ensemble des agents et la notation préfixée indique que l'on accède aux membres de l'instance sur laquelle l'appel est fait.

Algorithm 1 Algorithme d'ordonnancement de la simulation

Require: $T > 0$
1: $t \leftarrow 0$
2: **while** $t < T$ **do**
3: **for all** $a \in \Omega_{\mathcal{A}}$ **do**
4: $CategorieAccessible \leftarrow$
5: $perception(a.position, a.CategoriePertinente)$
6: $a.activate(CategorieAccessible)$
7: **end for**
8: $t \leftarrow t+1$
9: **end while**

Lorsqu'un agent est activé, il exécute sa boucle *perception - décision - action*. Notre proposition concerne l'étape *perception* pour laquelle une partie de la tâche a déjà été réalisée puisque l'agent a les descriptions accessibles et il doit trouver les contextes possibles. Le parcours de *CategorieAccessible* (algorithme 2) est déjà une sélection des filtres potentiels puisque les arbres dont la Catégorie référence n'est pas dans la liste sont ignorés. Figure 3, seuls les arbres de filtres des Catégories *VA* et *TS* sont explorés et pas l'arbre de la Catégorie *PA* grâce à la coupe numéro 1.

Dans l'algorithme 2, l'agent activé parcourt toutes les Catégories accessibles et pour chacune d'elle, il parcourt l'arbre des filtres la concernant (la notation $self$ référence l'agent). Les arbres de filtres sont enregistrés dans un dictionnaire ordonné, appelé $Filter$ avec le nom de la Catégorie comme clef et l'arbre de filtres comme valeur.

Ce parcours est en deux étapes. La première concerne l'exploration des filtres du nœud courant (valeur 1 dans l'algorithme 2-(2)) : la partie f_a (condition sur l'état de l'agent) du filtre est testée avant de tester f_C (autres conditions sur le contexte). Le respect de cet ordre évite de parcourir les entités concernées si l'état courant de l'agent rend le filtre inadapté. Par exemple, pour le filtre *warning* il est inutile de tester tous les agents véhicules perceptibles si l'agent n'est pas dans la voie centrale. Si l'agent utilise un automate de comportement, le concepteur peut mettre ainsi une condition sur l'état de l'automate à ce niveau de l'exploration. La seconde étape conerne l'exploration des enfants sauvegardés dans une sous-liste (valeur 2 dans l'algorithme 2-(9)) : l'exploration est réalisée selon un processus récursif et les mêmes principes que pour la racine.

Algorithm 2 Algorithme d'activation de l'agent

Require: $CategorieAccessible$
1: **for all** $categorie \in CategorieAccessible$ **do**
2: **for all** $f \in self.Filter[categorie][1]$ **do**
3: **if** $f.valid(self)$ **then**
4: **if** $f.trigger(self, CategorieAccessible)$ **then**
5: $self.validFilter.add(f)$
6: **end if**
7: **end if**
8: **end for**
9: **for all** $t \in self.Filter[categorie][2]$ **do**
10: $self.recursiveTriggering(t, CategorieAccessible)$
11: **end for**
12: **end for**
13: $self.decision()$
14: $self.action()$

Si le filtre testé est validé par l'état de l'agent (algorithme 2-(3)) et des descriptions nécessaires (algorithme 2-(4)), alors il est enregistré dans une liste des filtres valides. Cette liste est constituée de sous-listes contenant le nom du filtre et le n-uplet de descriptions le validant.

Ainsi le paramètre d'entrée est l'arbre à parcourir (*arbrePartiel*) qui est enregistré comme une liste de listes imbriquées avec pour chaque niveau d'imbrication trois informations : 1) le nom de la nouvelle Catégorie prise en compte ; 2) la liste des filtres du noeud ; 3) l'arbre des fils.

Algorithm 3 Exploration récursive des filtres

Require: *arbrePartiel*
Require: *CategorieAccessible*
1: **if** $arbrePartiel[1] \in CategorieAccessible$ **then**
2: **for all** $f \in arbrePartiel[2]$ **do**
3: **if** $f.valid(self)$ **then**
4: **if** $f.trigger(self, CategorieAccessible)$ **then**
5: $self.validFilter.add(f)$
6: **end if**
7: **end if**
8: **end for**
9: **for all** $t \in arbrePartiel[3]$ **do**
10: $self.recursiveTriggering(t, CategorieAccessible)$
11: **end for**
12: **end if**

Avec ces informations, si la Catégorie n'existe pas (algorithme 3-(1)) le parcours est arrêté sinon les filtres du nœuds (algorithme 3-(2)) sont testés ainsi que les nœuds fils (algorithme 3-(9)). Si la Catégorie n'appartient pas aux Catégories accessibles alors cette partie de l'arbre n'est pas explorée. Par exemple, les arbres de filtres contenant la Catégorie *PA* ne sont pas explorés car cette Catégorie n'appartient pas aux Catégories accessibles (figure 3-coupe 2).

6 Expérimentation

6.1 Configuration de la simulation

Afin de valider notre algorithme, nous avons choisi un cadre théorique afin de pouvoir maîtriser l'ensemble des paramètres qui peuvent expliquer l'évolution des résultats. Notre environnement est une grille 2D qui contient 135.000 entités distribuées en 6 Catégories auquel s'ajoute 100.000 agents. Pour chaque Catégorie, nous avons fixé le nombre d'entités (table 1) afin d'avoir des Catégories faiblement représentées ($C4$) ou bien représentées ($C9$). Pour chaque description, nous générons des valeurs aléatoires entre 0 et 20 pour 5 propriétés.

Dans chaque filtre, nous définissons deux conditions par Catégorie (f_C) et la partie relative aux tests de l'agent (f_a) comprend trois conditions. Par exemple, un filtre de profondeur 2 aura sept conditions (4 conditions pour f_C et trois pour f_a). Nous simulons des entités situés aléatoirement sur une grille dont la taille varie de 1000×1000 à 7000×7000. Les agents décident de l'action à réaliser à partir des informations issues de leur champ de perception.

Un filtre est constitué de boucles imbriquées et chacune concerne une Catégorie. Les boucles

TABLE 1 – Cardinalités des différentes Catégories

nom	*cardinalité*
$C4$	10000
$C5$	15000
$C6$	20000
$C7$	25000
$C8$	30000
$C9$	35000

imbriquées sont ordonnées selon l'ordre croissant des Catégories. Par exemple, pour le filtre $F2-78$, il y a deux boucles : la boucle de la Catégorie $C8$ est imbriquée dans la boucle de la Catégorie $C7$. Classiquement, l'objectif est de minimiser le coût d'évaluation du filtre en testant le minimum d'entités. En effet, lorsqu'une description valide les conditions de la boucle courante (pour $F2-78$, $\omega_i \in C7$), alors la boucle imbriquée est parcourue. A nouveau, si une description dans la boucle courante valide les conditions (pour $F2-78$, $\omega_j \in C8$) et qu'il n'y a plus de boucles imbriquées, alors le filtre est valide et le n-uplet d'entités (pour $F2-78$, $(\omega_i \in C7, \omega_j \in C8)$) et la fonction *trigger* retourne vraie (algorithme 2-(4) ; algorithme 3-(4)).

La liste des arbres de filtres est celle donnée par la figure 4. Les filtres choisis respectent une répartition homogène entre toutes les Catégories pour pas introduire de biais. Ainsi pour chaque Catégorie, il existe trois filtres de premier niveau (*F1-x*), six filtres de second niveau (*F2-x*) et un filtre de troisième niveau (*F3-x*) pour un total de 41 filtres pour chaque agent donc 4.100.000 filtres par pas de temps.

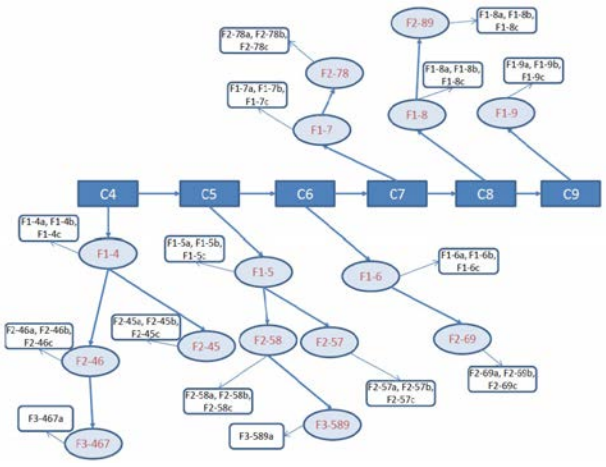

FIGURE 4 – Organisation générale des données

Nous comparons notre algorithme avec une suc-

cession de branchements conditionnels et non de branchements conditionnels imbriqués, ce qui représenterait un algorithme spécifique alors que notre structure est générique. Nous appelons cet algorithme *classique* alors que notre algorithme est appelé *structuré*. Le coût d'évaluation d'un filtre est identique dans chaque algorithme, seule l'organisation de leur recherche diffère. C'est pourquoi, nous choisissons de ne retenir que la première combinaison d'entités qui réussit. Le calcul des autres solutions ne permettrait pas d'évaluer notre algorithme mais les filtres eux-mêmes.

Nous réalisons des simulations de 30 cycles et mesurons le temps passé pour trouver les filtres possibles. Afin d'assurer un comportement similaire des deux algorithmes (même état du monde lors de l'évaluation), l'agent activé exécute à chaque cycle les deux algorithmes et mesure leur temps d'exécution avant de modifier l'état du monde. Pour cette dernière action, nous modifions la position de l'agent, ce qui permet à chaque nouveau cycle d'avoir un monde différent. Nos algorithmes ont été réalisés en Python 3.3 et exécuté sur un PC Intel Core i5-2500 CPU@3.3GHz avec 12 GB de mémoire.

6.2 Résultats

Nous proposons un algorithme qui optimise la détermination du contexte dans la phase de perception. Cette phase comprend également le parcours de la matrice contenant les données perçues par l'agent, ce qui est également un traitement coûteux. Il faut donc pouvoir évaluer le gain apporté par notre proposition par rapport au temps total nécessaire à la phase de perception. Nous proposons deux paramètres d'étude :

– Taille du champ de perception : la variation de ce paramètre permet de savoir quand la diminution du temps de calcul du contexte devient négligeable en comparaison au temps nécessaire pour explorer la matrice contenant les entités perceptibles.
– Taille de l'environnement : la variation de ce paramètre permet de modifier le nombre potentiel d'entités perçues avec un coût constant d'exploration de la matrice perçue.

Notre premier résultat concerne le pourcentage que le calcul du contexte représente sur la totalité du temps pris par la phase de perception pour l'algorithme *classique*. Les résultats sont données dans le tableau 2. Par exemple, si la valeur du champ de perception est 10 et la taille de l'environnement est 5000×5000 alors 29.07% du temps global de la phase de perception concerne le calcul du contexte et donc 70.93% concerne le parcours de la grille perçue par les agents. Nous observons que la détermination du contexte représente la moitié du temps de la phase de perception lorsque le champ de perception est petit et que cette valeur décroît rapidement avec l'augmentation du champ de perception (baisse à 17.53% pour une matrice de 7000×7000 et un champ de perception de 20). Si le champ de perception est plus grand que 20 alors le temps d'exécution concernant la détermination du contexte devient négligeable par rapport au temps nécessaire pour parcourir la matrice. L'augmentation de la taille de la grille fait baisser ce pourcentage car le temps d'exploration demeure stable mais le temps de calcul de la détermination du contexte décroît car il y a moins d'entités perçues par chaque agent.

TABLE 2 – Coût relatif du calcul de contexte par rapport à la phase de perception

	Champ de perception		
Taille de la grille	5	10	20
1000×1000	49.03%	35.52%	20.23%
3000×3000	38.95%	35.31%	24.96%
5000×5000	37.97%	29.07%	21.51%
7000×7000	35.47%	25.75%	17.53%

Le deuxième résultat est une comparaison des temps d'exécution des algorithmes *structuré* et *classique*. Le tableau 3 donne le pourcentage d'amélioration de notre algorithme selon la taille du champ de perception et la taille de la grille. Par exemple, si le champ de perception est de 10 et que la taille de la grille est 5000×5000, alors le temps nécessaire pour déterminer le contexte est de 48.1% moindre avec l'algorithme *structuré* qu'avec l'algorithme *classique*.

TABLE 3 – Performance relative du coût du calcul de contexte

	Champ de perception		
Taille de la grille	5	10	20
1000×1000	8.11%	13.93%	8.46%
3000×3000	57.53%	20.85%	8.11%
5000×5000	82.35%	48.1%	10.66%
7000×7000	87.77%	69.04%	29.24%

Notre algorithme est toujours meilleur que l'algorithme classique. Cet avantage diminue avec l'augmentation de la taille du champ de percep-

tion et augmente avec l'augmentation de la taille de l'environnement. Ce résultat illustre l'objectif de notre algorithme qui est d'exploiter la nature des données perçues. En effet, plus il y a d'entités perçues (augmentation de la taille du champ de perception) moins il y a de Catégories ignorés. Néanmoins, nous avons vu dans la première série d'expérimentations que la taille du champ de perception doit rester limitée car avec une valeur importante, le coût de détermination du contexte devient négligeable par rapport au temps nécessaire pour identifier les entités perceptibles.

Le tableau 4 montre que notre algorithme diminue le temps d'exécution de la phase de perception quelle que soit la taille du champ de perception et donc le nombre d'entités perçues. Par exemple, si la taille du champ de perception est de 10 et la taille de l'environnement est de 5000×5000, alors le temps pour la phase de perception diminue de 13.98% avec l'algorithme *structuré* par rapport à l'algorithme *classique*. Par pas de temps et sur l'ensemble des configurations,, le gain maximum est de 1.82 seconde, le minimum de 0.51 seconde et le gain moyen est de 1.15 seconde.

TABLE 4 – Performance relative du temps de simulation

	Champ de perception		
Taille de la grille	5	10	20
1000×1000	3.98%	4.95%	1.71%
3000×3000	22.41%	7.36%	2.02%
5000×5000	31.27%	13.98%	2.29%
7000×7000	31.13%	17.78%	5.13%

Notre algorithme est dépendant de l'ordre des cardinalités des Catégories fourni par le concepteur. Afin de tester les conséquences d'un mauvais ordre, nous exécutons nos simulations avec la même liste d'arbres de filtres (figure 4) mais avec un ordre inverse des cardinalités (par exemple, nous avons 35.000 entités de la Catégorie $C4$ et 10.000 entités de la Catégorie $C9$). Les améliorations de l'algorithme *structuré* comparé à l'algorithme *classique* avec un mauvais ordre sont données dans le tableau 5. Nous constatons que l'algorithme *structuré* reste meilleur.

Notre approche déclarative par une formalisation du contexte par des filtres et l'utilisation d'un algorithme utilisant la structure de listes d'arbres permet d'associer à chaque agent sa propre structure sans avoir à modifier l'algorithme de détermination du contexte. Nous testons à présent les conséquences en termes de temps d'exécution. Nous avons réalisé des simulations avec des environnements de 500×500 à 5000×5000 avec 120.000 agents qui ont la liste d'arbres de filtres représentés figure 4 et trois Catégories (C1,C2,C3) de 40.000 agents qui n'ont qu'une sous partie de cette structure organisée comme décrit section 4 et contenant les Catégories selon la répartition suivante : $C1$ est concernée par $C4, C5, C8, C9$; $C2$ est concernée par $C6, C7, C8, C9$; $C3$ est concernée par $C4, C5, C6, C7$.

TABLE 5 – Performance relative du coût du calcul de contexte (mauvais ordre)

	Champ de perception		
Taille de la grille	5	10	20
500×500	11.88%	7.36%	1.4%
1000×1000	26.55%	12.39%	7.63%
3000×3000	76.78%	43.76%	18.31%
5000×5000	88.67%	66.98%	28.05%

La segmentation des centres d'intérêts des agents permet une diminution du temps de calcul pour les deux algorithmes puisque les entités prises en compte par chaque agent sont moins nombreuses. Notre algorithme confirme avec la segmentation qu'il est plus efficace que l'algorithme *classique*. Avec notre algorithme, le gain moyen du temps d'exécution d'un cycle avec la segmentation est de 7.61 seconde sur le temps sans segmentation.

7 Conclusion et perspectives

Dans cet article, nous avons proposé une solution pour diminuer le temps d'exécution d'une simulation multi-agent tout en permettant une modélisation de contextes complexes. Cette problématique est importante car comme tout modèle de micro-simulation, les SMA rencontre des difficultés de passage à l'échelle à cause de leur coût d'exécution.

Notre proposition s'est focalisée sur la problématique du coût de calcul du contexte. Nous proposons de modéliser un contexte comme un filtre et pouvons ainsi proposer une structure et un algorithme pour traiter au mieux cette partie de l'activité de chaque agent activé. Notre structure exploite la cardinalité des différents Catégories de composants qu'un agent peut prendre

en compte dans l'évaluation de son contexte. Cette structure est simple et ne prend pas en compte la structure des filtres comme peut le faire un algorithme comme RETE [9]. L'avantage est que son coût mémoire est limité à l'enregistrement des filtres et n'est pas dépendant de la dynamique de l'environnement comme cela serait le cas avec RETE. Nous devons à présent examiner de nouvelles structures de données, comme les treillis, afin d'organiser les filtres selon différents critères de classification. Nous avons constaté que l'amélioration que nous apportions devient négligeable face au coût d'accès aux données (concrétisé par un champ de perception important). Une amélioration consisterait donc à intégrer les travaux pour optimiser l'accès aux données comme ceux proposés dans [13, 7]. De plus, nous devons continuer notre évaluation en testant notre algorithme avec différentes applications.

Références

[1] F. Badeig and F. Balbo. Définition d'un cadre de conception et d'exécution pour la simulation multi-agent. *Revue d'Intelligence Artificielle*, 26(3) :255–280, 2012.

[2] F. Béhé, S. Galland, N. Gaud, C. Nicolle, and A. Koukam. An ontology-based metamodel for multiagent-based simulations. *Simulation Modelling Practice and Theory*, 40(0) :64 – 85, 2014.

[3] F. Bousquet, I. Bakam, H. Proton, and C. L. Page. CORMAS : Common resources and multi-agent systems. In Springer-Verlag, editor, *Tasks and Methods in Applied Artificial Intelligence*, pages 826–837, 1998.

[4] N. Collier. Repast : An extensible framework for agent simulation. *The University of Chicago's Social Science Research*, 36, 2003.

[5] A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer, 1999.

[6] R. Evertsz, F. E. Ritter, P. Busetta, M. Pedrotti, and J. L. Bittner. Cojack-achieving principled behaviour variation in a moderated cognitive architecture. In *Proceedings of the 17th conference on behavior representation in modeling and simulation*, pages 80–89, 2008.

[7] N. Farenc, R. Boulic, and D. Thalmann. An informed environment dedicated to the simulation of virtual humans in urban context. In *Proceedings of EUROGRAPHICS 99*, pages 309–318, 1999.

[8] J. Ferber and O. Gutknecht. Madkit : A generic multi-agent platform. In *4th International Conference on Autonomous Agents*, pages 78–79, 2000.

[9] C. L. Forgy. Rete : A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19 :17–37, 1982.

[10] Y. Kubera, P. Mathieu, and S. Picault. Interaction-oriented agent simulations : From theory to implementation. In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of ECAI'08*, pages 383–387. IOSPress, 2008.

[11] S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan. Mason : A new multi-agent simulation toolkit. In *SwarmFest Workshop*, 2004.

[12] F. Michel. The irm4s model : the influence/reaction principle for multiagent based simulation. In *AAMAS*, volume 7, pages 1–3, 2007.

[13] F. Michel. Intégration du calcul sur gpu dans la plate-forme de simulation multi-agent générique turtlekit 3. In S. Hassas and M. Morge, editors, *JFSMA*, pages 189–198. Cepadues Editions, 2013.

[14] M. Sierhuis, W. J. Clancey, and R. J. Van Hoof. Brahms : a multi-agent modelling environment for simulating work processes and practices. *International Journal of Simulation and Process Modelling*, 3(3) :134–152, 2007.

[15] G. Wagner. Aor modelling and simulation : Towards a general architecture for agent-based discrete event simulation. In *Agent-Oriented Information Systems*, pages 174–188. Springer, 2004.

[16] T. Warden, R. Porzel, J. D. Gehrke, O. Herzog, H. Langer, and R. Malaka. Towards ontology-based multiagent simulations : The plasma approach. In *24th European conference on modelling and simulation (ECMS 2010)*, pages 50–56, 2010.