A Tree-Based Context Model to Optimize Multiagent Simulation

Flavien Balbo[1], Mahdi Zargayouna[2], and Fabien Badeig[1]

[1] Institut Henri Fayol, ENS Mines Saint-Etienne,
158 Cours Fauriel, 42100 Saint-Etienne, France
{flavien.balbo,fabien.badeig}@mines-stetienne.fr
[2] Université Paris-Est, IFSTTAR, GRETTIA,
Champs sur Marne, France
hamza-mahdi.zargayouna@ifsttar.fr

Abstract. In most multiagent-based simulation (MABS) frameworks, a scheduler activates the agents who compute their context and decide the action to execute. This context computation by the agents is executed based on information about themselves, the other agents and the objects of the environment that are accessible to them. The issue here is the identification of the information subsets that are relevant for each agent. This process is time-consuming and is one of the barriers to increased use of MABS for large simulations. Moreover, this process is hidden in the agent behavior and no algorithm has been designed to decrease its cost. We propose a new context model where each subset of information identifying a context is formalized by a so called "filter" and where the filters are clustered in ordered trees. Based on this context model, we also propose an algorithm to find efficiently for each agent their filters following their perceptible information. The agents receive perceptible information, execute our algorithm to know their context and decide which action to execute. Our algorithm is compared to a "classic" one, where the context identification uses no special data structure. Promising results are presented and discussed.

Keywords: Multiagent simulation, Context, Agent models.

1 Introduction

One of the main functional objectives of the simulation domain is the controlled reproduction of complex systems. The simultaneity of actions, which means that several agents are activated at the same simulated time, is one of the properties that must be ensured. Therefore, the execution of a MABS model enforces a scheduling process (executed by a scheduler) that synchronizes the agents execution and simulates the simultaneity of their behaviors. Most of the MABS frameworks follows a cooperative model, where the activation of agents is controlled by a scheduler and their interruption is controlled by the agents themselves. When activated by the scheduler, the agent executes his current behavior and decides when to hand over to the scheduler.

When the agent is activated in a cooperative model, he is aware of the state of the simulation and his action will change that state. However, the agent takes simultaneously into account all the information accessible to him [5] and this could imply important computation times. The issue for the agent is to find the subsets of information

J.P. Mller, M. Weyrich, and A.L.C. Bazzan (Eds.): MATES 2014, LNAI 8732, pp. 251–265, 2014.

that are relevant for him. These subsets, that we call context, condition his behavior and belong to the his internal knowledge. In most MABS, the relevance criterion of a context is embedded in the agent implementation and is not separated from the actual action to execute. It is therefore difficult to customize the context computation without a modifying the agent's implementation. To decrease the context computation cost, designers often use nested contexts and/or behavioral automaton (NetLogo-like MABS). From our point of view, nested contexts make it hard to design agents and to customize context modeling. Behavioral automata on the other hand, by focusing on the internal state of the agent, neglect the other components of the agent context. Our proposal is the modeling of the contexts as "filters" to simplify the agent design without limiting the context computation possibilities. The activated agent receives his perceptible information from the scheduler and executes our algorithm to find the filters associated with his current context.

The remainder of the paper is organized as follows. Section 2 discusses the issues related to context computation. Section 3 presents an illustrative example that is followed all along this paper. Section 4 provides the formal definition of our proposal. Our context selection algorithm is provided in Section 5. Section 6 presents our experimentation and results. The paper concludes with a discussion and some perspectives to this work.

2 State of Art

When an agent is activated, he is aware that he is executing a new simulation time step. He can therefore compute his current context before to decide which action to execute. In this section, we discuss the options to compute the agent's context.

The first option, the most popular, is agent-oriented. The scheduler activates the agents either by calling a default method [3,13] or with a control message [11,14] and the activated agent computes his context. For instance in [3] the objects belonging to the perception field of the activated agent are given to him with a perception event. The logo-based multiagent platforms such as the TurtleKit simulation tool of MADKIT [7] or STARLOGO[1] have chosen this option. The agent is activated following the state of his behavioral automaton that has been computed at the previous activation.

The second option is scheduler-oriented. The scheduler computes for the agents which action to execute following their current context. To the best of our knowledge, the framework JEDI [9], the Repast Simphony simulation platform [4] and our own work [1,2] are the only proposals where the choice of the action that is executed by an agent is computed by the scheduler. In the JEDI framework [9], the choice of an action by the agent is based on an interaction matrix where a cell is a conditioned contextual interaction between two agents. For instance, an interaction is possible between two agents following their proximity. To each of these contexts, an action is associated and will be executed by the activated agent. This interaction matrix is defined by the designer and does not change during the simulation. Repast Simphony natively uses the first scheduling options (i.e. with a default method), but it also allows a sort of *contextual activation* based on "watchers". Watchers allow an agent to be notified of a state

[1] `http://education.mit.edu/StarLogo/`

change in another agent and schedule the resulting action. The designer specifies which agent to watch and a query condition that must be verified to trigger the resulting action. This activation process is limited by the expressiveness of the watcher queries language to express the activation context. The queries are boolean expressions that evaluate the watcher and the watchee using primitives such as *colocated*, *linked_to* [*network name*], *within X* [*network name*], etc. (a network is a graph of agents relationships) and the operators AND and OR. It is not possible to integrate complex conditions about other components (other than the watcher and the watchee).

In previous works [1,2], we have proposed a multiagent-based simulation process that belongs to this last option. We have modeled contexts with conditions about shared information on the MAS components. A subset of conditions defines a specific context and is called a filter. The multiagent environment is used as a scheduler and it activates the agents according to filters triggering based on perceptible information. In the proposal described in this paper, the environment activates the agents in turn with their accessible information and it is the activated agent who computes his context based on his own context model (his filters).

3 Illustrative Example

To illustrate our proposal, we present an example of context modeling for a driver entering a roundabout. This example illustrates the components of our proposal and our experiments are based on a theoretical example (Section 6). Figure 1 represents a roundabout with agents (vehicles, pedestrians and bicycles) and objects (traffic signs).

In our proposal, an agent context is a conditioned combination of the perceptible information that are relevant for him. The only perception of the information is not sufficient, their values have also to be taken into account. For instance, the pedestrian agent pa_1 is perceived by va_1 and va_2 (Figure 1) but the resulting context is not the

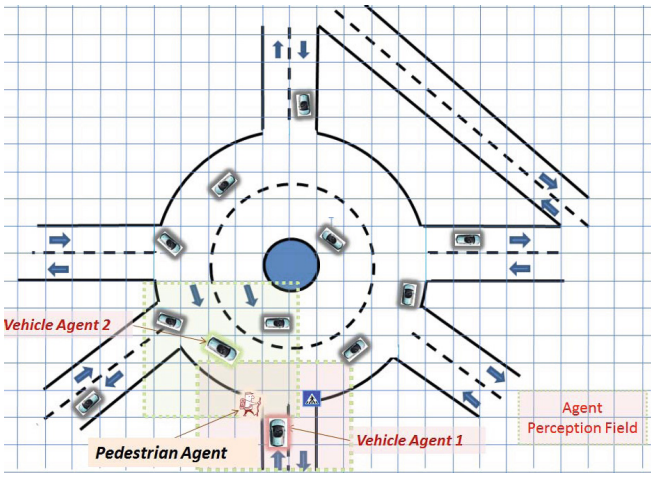

Fig. 1. Roundabout simulation example

same for each agent: 1) va_1 context could be *"my speed is excessive and there is a pa_1 crossing the street before me"*; 2) va_2 context could be *"I am about to cross entering traffic, which is blocked by the crossing pa_1"*. The information are indeed the same, but it is their combination that is relevant for the agents. After identifying their contexts, the agents have to decide which action to execute. The issue is that these combinations are multiple and their computation is time-consuming. The objective is to decrease the context computation time without limiting the expressiveness of the context definition. This definition is related to the domain expert, as well as their use in the decision process. Moreover, the design of the agents using context information remains free. For instance, va_1 could be a BDI agent who would have initiated a plan with this information and v_2 could be a reactive agent who would reacted with an acceleration. Our proposal is placed between the information acquisition and the decision process: we propose a data model and an algorithm to process information context.

4 Context Model Definition

Context computation assumes that agents have information about the MAS components (agents, objects, etc.) that are accessible to them. The accessibility conditions have to be specified for each simulation and, in our example, we associate to each agent a perception field where all simulation components are perceptible by him. In this section, we propose a context model. The first component of the model is called an entity and is a meta-information about a MAS component.

Definition 1 (Entity). *An* entity $\omega \in \Omega$ *is a* $\langle r_\omega, d_\omega \rangle$ *pair with :*

- r_ω*: reference to a real component of the the MAS, i.e. agent or object.*
- d_ω*: description of this component recorded in the environment. It is defined by a set of* $\langle property, value \rangle$ *pairs.*

r_ω gives access to the component (for the activation process if it is an agent); d_ω contains information to identify the context of the agents. An entity is the link between the MAS and the context model. In the following (except where noted), entity and description are used interchangeably. A property gives a specific information about a component of the MAS.

Definition 2 (Property). *A property $p_i \in \mathcal{P}$ is a function, which description domain $d_j \in \mathcal{D}$ is quantitative, qualitative or a finite set of data. A property is noted $p_i : \Omega \rightarrow d_j$, with Ω the set of descriptions.*

The properties are used to characterize subsets of entities.

Definition 3 (PDescription). *A PDescription is a subset of $\mathcal{P}$ and we note P_e the PDescription of the entity e.*

The extension of a *PDescription* is called a *Category*.

Definition 1 (*Category*). *A Category is a subset of semantically similar entities with the same PDescription :* $\langle label, \{\omega \in \Omega | P_{\omega_i} = P_{\omega_j} \forall \omega_i, \omega_j \in C_x\} \rangle$ *with label the name of the Category.*

In our example, vehicle agents (VA), pedestrian agents (PA) and traffic signs (*TS*) are category examples. At least, the list of the perceptible entities of an agent is given to him by environment at each time step. This list is called *PerceptibleCategories* and does not contain empty categories. For instance, the description of a vehicle agent could be (we note $\Omega_{\mathcal{A}} \subset \Omega$ the set of agents):

- $speed : \Omega_{\mathcal{A}} \rightarrow \mathbb{R}$: the speed of the agent;
- $location : \Omega_{\mathcal{A}} \rightarrow \mathbb{N}$: the distance from roundabout entry or a relative value for a roundabout lane;
- $street : \Omega_{\mathcal{A}} \rightarrow \{street\ name, lanelocation\}$: the name of the street or the location in the roundabout;
- $direction : \Omega_{\mathcal{A}} \rightarrow \{$ *to roundabout, from roundabout*$\}$;
- $turnSignal : \Omega_{\mathcal{A}} \rightarrow \{left, right, off\}$: the state of the turn signals;
- ...

Its $PDescription$ is $\{speed, location, \dots\}$ and its Category is $\langle vehicle, \{va_1, va_2, \dots\}\rangle$. We propose to model a context as a filter, which tests the entity that the agent perceives. A filter generates processed information from raw information (description of the MAS components).

Definition 4 (Filter). *A filter $F_j \in \mathcal{F}$ is a tuple $F_j = \langle f_a, f_C, n_f\rangle$ with:*

- $f_a : \Omega_{\mathcal{A}} \rightarrow \{true, false\}$ *a mandatory assertion that expresses constraints on the agent who owns the filter;*
- $f_C : 2^{\Omega} \rightarrow \{true, false\}$ *an optional set of assertions expressing constraints on others components that complete the context;*
- n_f *the filter name.*

A filter identifies by unification the agent's description and the context (subset of entities) that matches the associated assertions. A filter is valid for tuples $\langle a \in \Omega_{\mathcal{A}}, context \subset \Omega\rangle$ such that $f_a(agent) \wedge f_C(context)$ is evaluated to true. When a filter is valid, the associated context, $\langle context, n_f\rangle$, is valid for the agent a. A context being formalized as assertions on the descriptions of the MAS components, the $context$ and n_f information are complementary to characterize the MAS context. It means that the same description's subset can valid several contexts and a context can be validated by several description's subsets.

Let $\langle f_a, f_C, warning\rangle$ be a filter dealing with the detection of a warning related to the potential movement of vehicles. A filter belongs to an agent and is therefore built from his point of view. For the *warning* filter, the vehicle agent is on the central lane of the roundabout (fig 1) and a slower vehicle agent before him in the other lane turns on his left turn signal. The filter triggering depends on: i) the location of the agent (assertion f_a), ii) the perception of another agent with a perceptible property (assertion f_C). The filter *warning* has the following definition:

- $a \in \Omega_{\mathcal{A}}$: $f_a : [speed(a) = ?s_a] \wedge [street(a) = centralLane] \wedge [location(a) = ?l_a]$
- $b \in \Omega_{\mathcal{A}}$: $f_C(b) : [speed(b) < ?s_a] \wedge [location(b) < ?l_a + 2] \wedge [turnSignal(b) = left] \wedge [street(b) = externLane]$

The symbol "?" before an expression identifies a variable and the operator "=" is the comparison operator. With this filter, the agent a, when he is in the central lane, is aware of the b slower agents who are in the external lane and up to two units before him, with their left turn signal on. The scheduler has already filtered the perceptible entities based on the perception field of the agent a.

We assume that the number of entities by category (the categories' cardinality) is defined by the MABS designer, or that he is at least able to order categories following their cardinalities (without necessarily defining the exact number of entities per category). In our roundabout example, the designer does not know the exact number of entities but he is able to define the following order $\cdots < |TS| < |PA| < |VA| < \ldots$ if the simulation concerns rush hours with a great traffic activity, or $\cdots < |PA| < |VA| < |TS| < \ldots$ if the simulation concerns night time with low traffic. The rank of the filters will follow the chosen order.

Using this order, the context-knowledge of the agents is formalized as an ordered list of pairs, that we call *PotentialContext*. The first member of a pair is the label of a category, which is called *reference*, and the second member is an ordered tree of filters. This tree contains the filter with *reference* as the tested category with the minimal cardinality. For instance, Figure 2 describes the pair (TS, tree) for a vehicle agent. The category TS is associated to a tree containing the filters where the category TS is tested alone or with the categories PA and VA. We associate with the category PA (the second element of the list) the filters where the category PA is tested alone or along with the category VA. At least, we rank the *PotentialContext* list following the cardinality of the reference.

Each agent has his own customized instance of the proposed data structure and he processes our algorithm (described later in this paper) to browse it and find the filters

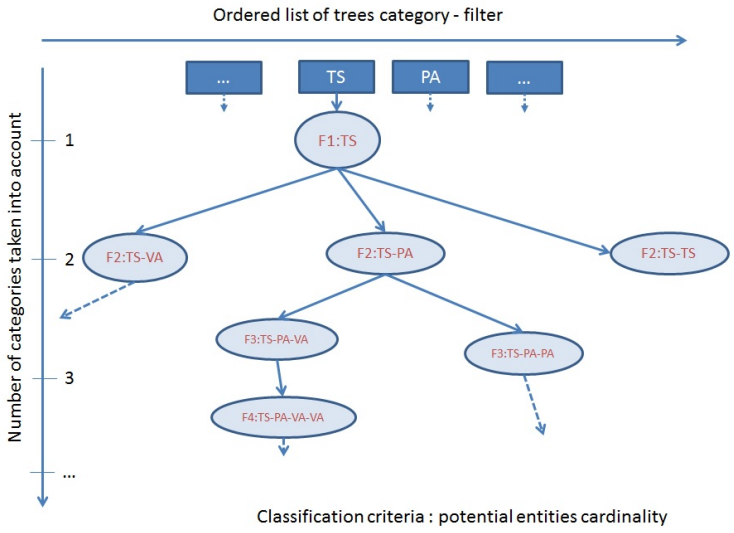

Fig. 2. Ordered list of filters

that match the accessible descriptions. The objective of this structure is to test the less filters given the number of categories that are perceived by the agent. The basic idea is the following: *The evaluation of a filter is conditioned by the existence of an entity for each category that it tests.*

The naming convention of the filters indicates the depth of the filters in the tree and the increasing cardinality-order of the categories to test. For instance *F2:TS-VA* (Figure 2) is a filter that is at the depth 2 and testing the categories *TS* then *VA*. The cardinality-order has two advantages. The first is algorithmic because it allows us to look efficiently for filters that can be evaluated (Section 5). The second advantage is practical since it insures the uniqueness of filters in the tree. For instance, the filter *F2:TS-VA* does not belong to the tree of the category *VA*. Nevertheless, for clarity's sake, when the position of the filter in the tree is not discussed, we use a more explicit naming, as for the *warning* filter defined earlier for the filter *F1:VAx*.

A node is a set of filters for which f_C validation concerns the same set of categories. To distinguish them, we append a letter to the end of the filters name. For instance, the node *F4:PA-TS-VA-VA* (Figure 2) contains all filters where f_C is validated with the description of a pedestrian agent, a traffic sign and two vehicle agents (in addition to the vehicle agent, owner of the list of filters' tree). The filter *warning* belongs to the node *F1:VA* since f_C is related to one vehicle agent.

An arc is an inclusion relation between subsets of filters: the deeper node (the child) contains the filters for which evaluation needs one more category to test than the shallower node (the parent). For instance, the children of the node *F1:TS* are *F1:TS-VA*, *F1:TS-PA* and *F1:TS-TS* with respectively the addition of the categories *VA, PA* and *TS*.

For a given depth, the filters are ranked in decreasing order of categories cardinality. For a given node, these children are explored if the additional category belongs to the perceptible categories (Section 5). Therefore, processing in priority the children that have potentially the more chances to have descriptions increases the possibility to have a valid context and to stop the search. For instance, there are potentially more vehicle agents than pedestrian agents that are perceived by a vehicle agent. In Figure 2, the filters belonging to the node *F3:TS-PA-VA* are tested before the filters belonging to the node *F3:TS-PA-PA* if the category *VA* belongs to *PerceptibleCategories*. However, if the objective is to retrieve all the possible contexts of the agents then the ordering of the nodes has no consequence.

If a child node has no parent, i.e. there exists no filter concerning only the parent's categories, then the parent node is created but is empty.

Starting from this structure of filters and the perceptible descriptions, we can design an algorithm that identifies efficiently the possible filters.

5 Context Computation Algorithm

The general principle is to test the only filters for which there exists descriptions that are accessible to the agent. For each reference, the agent has to test the filters contained in the root then in each of its children if the added category exists in *PerceptibleCategories*. It is noteworthy that a child node may have validated filters because accessible descriptions validate its conditions while its parent does not contain

any valid filter. Our context model exploits the structure of the perceived information, the category, and not their value. The advantage is the independence of our proposal from the environment dynamics, avoiding costly updates of the agent knowledge.

In the scheduling Algorithm 1, the number of time ticks is fixed (T) and for each tick, the scheduler activates in turn the agents and provides a list *PerceptibleCategories* to each of them. This list is built by environment, which selects among perceptible entities the ones that are relevant for the activated agent (1-(5)). A category is relevant if it is related to at least one filter. The list *RelevantCategories* is defined for each agent as the list of the references of his relevant categories. This list is not sorted because our algorithm aims to provide all the possible contexts; it is then necessary to explore all the possible trees. The prefixed notation indicates the access to the members of the concerned element and we note $\mathcal{A}$ the set of agents.

When an agent is activated, he executes a *perception* - *decision* - *action* loop. A part of the perception step is already performed since the agent has the perceptible entities. The browsing of *PerceptibleCategories* (Algorithm 2) is already a selection of the filters, because if a category refers to a filter's tree ft but does not belong to *PerceptibleCategories*, ft is not explored. In Figure 3, only the list of filters' trees of the categories *VA* and *TS* are explored and not the filters' tree of the category *PA* following the selection labeled with the number 1.

In Algorithm 2, for each category belonging to *PerceptiblesCategories*, the agent explores the related filters' tree.

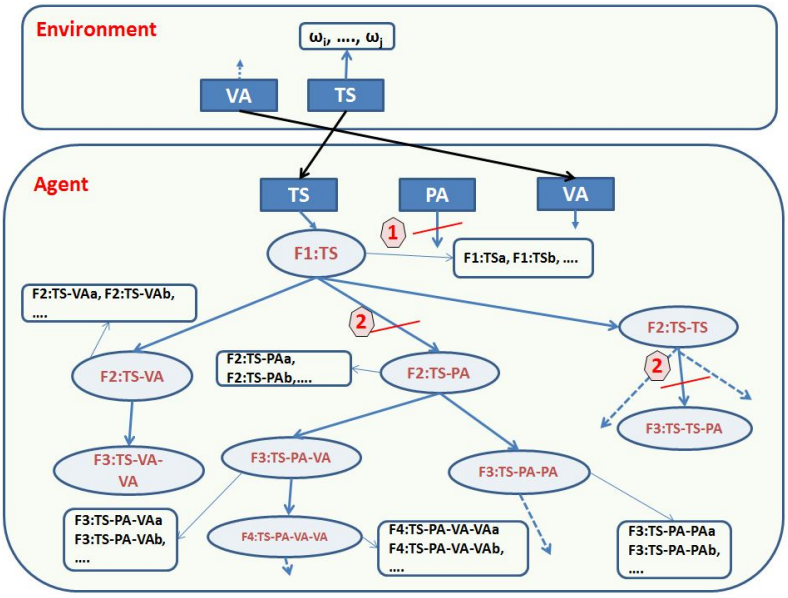

Fig. 3. Global overview of context model

Algorithm 1. Simulation scheduling algorithm

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

The filter's trees (*PotentialContext*) are recorded in an ordered dictionary with the category name as a key and the filters' trees as value. The algorithm explores the filter's tree in two steps:

1. It explores the filters of the current node (value 1 in Algorithm 2-(2)): it tests the f_a part of the filter (condition on the state of the agent) before to test f_C (the conditions on the concerned descriptions). This order avoids to browse the related categories if the current state of the agent makes the filter not adapted. For instance, for the filter *warning*, it is not useful to test all the perceptible vehicle agents if the activated vehicle agent is not in the central lane of the roundabout. If the agent uses a behavioral automaton, his current state can be used here to reproduce a logo-based simulation.
2. It explores the children saved in a sublist (value 2 in Algorithm 2-(9)): the exploration of the child is performed following a recursive process applying the same principles than for the root.

If the filter is valid given the state of the agent (Algorithm 2-(3)) and the necessary descriptions (Algorithm 2-(4)) then it is saved in the list of valid filters of the agent.

Algorithm 2. Activate: Agent activation algorithm

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

This list is made of sublists containing the name of the filter and the list of descriptions validating it. We choose not to compute all the combinations of perceptible descriptions of a given filter and to only select the first successful.

The input parameters of the recursive algorithm are the part of the filters' tree that is explored and the accessible descriptions. A partial filters' tree is a list of lists with, for each imbrication level, three information:

1. The name of the new category taken into account. For instance *VA* for the fist call following the filters tree given Figure 3.
2. The list of filters of the node. For instance the filters belonging to the node *F2:TS-VA* Figure 3
3. The list of children that reproduce this structure. For instance the structure related to the filters' tree with *F3:TS-VA-VA* as a root.

With these information, the algorithm tests the existence of the category (Algorithm 3-(1)) and if successful, it tests the nodes of the filter (Algorithm 3-(2)) then accesses the children nodes (Algorithm 3-(7)). If the category does not belong to perceptible categories then this part of the filters' tree is not explored. For instance, the filters' tree with the category *PA* (Figure 3-selection labeled with 2) are not explored because this category does not belong to perceptible categories.

6 Experimentation

To validate our proposal, we choose a theoretical framework in which we set categories and filters. Our environment is a 2D grid that contains 135,000 entities distributed in 6 Categories (C4 to C9) in addition of 100,000 agents (category C1). For each Category, we set a relative number of entities to have poorly represented categories ($C4$) or well represented categories ($C9$) (Table 1). For each description, random values between 0 and 20 for five properties are generated.

Algorithm 3. Recursive tree of filters exploration

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

We simulate agents situated on a matrix with a size varying from 1000×1000 to 7000×7000. The agents must decide which action to perform following the MAS entities that are present in their perception field. The position of the entities is random.

Table 1. Cardinality of the Categories

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

The filters' tree for our tests is the one described in Figure 4. Filters are chosen to respect a homogeneous dispatching between categories in order not to introduce bias. An obvious bias is the overrepresentation of filters for an underrepresented category. Hence for each category, there exists 3 filters of first level (F1), 6 filters of second level (F2) and a filter of third level (F3) for a total of 41 filters. Each agent of the simulation has the same filters' tree there is therefore 4,100,000 filters to test at each simulation step.

We compare our proposal with a solution in which filters are not organized and are explored iteratively. The objective is to compare our proposal with an algorithm computing the context with conditional branching but that remains generic. We call this proposal a *classic* algorithm while ours is called *structured* algorithm. The computation cost of a filter is similar in the two algorithms only the search organization is different.

We perform 30 simulations of one time cycle and measure the time spent to generate the possible filters. To ensure a similar behavior of the two algorithms (same world state during evaluation), at each cycle the activated agent executes both algorithms and their computation time is measured before modifying the state of the world.

Our algorithm have been developed in Python 3.3 and processed on a PC with an Intel Core i5-2500 CPU@3.3GHz and 12 GB memory.

We present an algorithm to reduce the context computation time during the perception step. Nevertheless this step includes the browsing of the grid containing perceptible entities which is also a costly computation. Therefore we must assess the advantage of our proposal according to the global computation time of the perception step. We propose two parameters for the evaluation:

- The size of the perception field: the variation of this parameter enables to know when the decrease of the context computation runtime becomes negligible according to the time needed to explore the grid that contains the perceptible descriptions.
- The size of the grid: the variation of this parameter enables to modify the number of potential entities that are perceived by the agent with a constant grid exploration cost.

The first result is about the percentage that the context computation process represents within the global perception step for the classic algorithm. The results are given in Table 2. For instance, if the perception field value is 10 and the size of the 2D grid is 5000×5000 then 29.07% of the perception step execution time is related to the context computation and therefore 70.93% to the browsing of the 2D grid that is perceived by each agent. We observe that the context computation represents half the execution time of the perception process when the perception field is small and it decreases quickly (down to 17.53% for a 7000×7000 grid and a perception field of 20). If the perception field is greater than 20 then the runtime related to the context computation process becomes negligible. The increase of grid size causes a decrease of the context computation time because the time to explore the grid remains stable while the context computation time decreases (there are less entities to process).

The second result concerns a comparison between the structured algorithm and a classic algorithm w.r.t. the context computation time. Table 3 provides the improvement percentage when using our algorithm w.r.t the perception field and the size of the grid. It means that if the perception field value is 10 and the size of the 2D grid is 5000×5000 then the necessary time to compute the context is 48.1% less with the structured algorithm than with the classic algorithm.

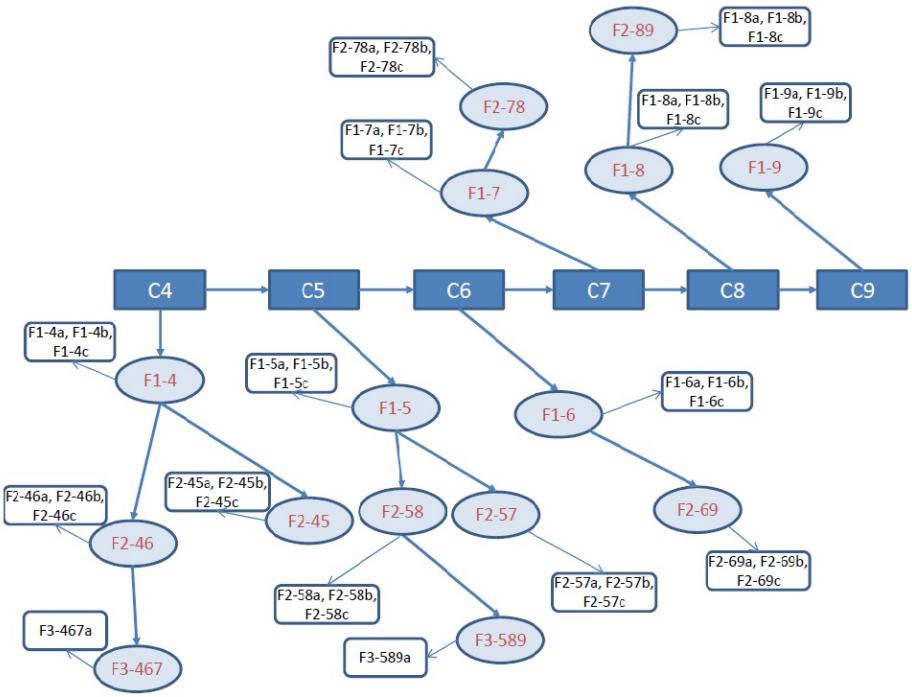

Fig. 4. List of trees of filters example

Table 2. Structured algoritm : Relative performance of context computation

	Perception field		
matrix size	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%

Our algorithm is always better than the classic algorithm but this advantage decreases conversely to the increase of the perception field. The decrease in the improvement with the increase of perception field is coherent with the principle of the algorithm. Indeed, the more entities the agent perceives, the less there are empty categories. Nevertheless, we have seen with the first experiment that the perception field has to be limited to 20, because with a superior value, the context computation time becomes negligible according to the browsing of the grid that contains the perceptible entities.

Table 4 highlights the fact that our algorithm improves the simulation execution time whatever the perception field size. For instance if the perception field value is 10 and the size of the 2D grid is 5000×5000 then the time related to the perception step is 13.98% less with the structured algorithm than with the classic algorithm. For each simulation step and for all simulation configurations, the maximal gain is 1.82 second, the minimal gain is 0.52 second and the average gain is 1.15 second.

Our choice to separate the context modeling and the algorithm to determine current context allows to give to each agent his own list of filters trees. This customization of the context can be processed without modifying the implementation of the agents. We perform simulations where the size of the environment evolves from 500×500 to 3000×3000 with 120,000 agents with the same list of filters (Figure 4) that we compare with three agents' categories (C1,C2,C3) with 40,000 agents each. Theses categories have a subset of the list of trees of filters. Each of these subsets contains the categories given in Table 5.

The context computation time decreases for the two algorithms because agents take into account less information to compute their context. Nonetheless, our algorithm remains always better than the *classic* algorithm. With our algorithm, the average decrease of the time is 7.61 seconds by cycle in comparison with the execution without the customization of the agents' context.

Table 3. Structured algorithm vs classic algorithm : Relative performance of context computation

	Perception field		
Matrix size	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%

Table 4. Structured algorithm vs classic algorithm : Relative performance of context computation

	Perception Field		
Matrix size	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%

Table 5. Dispatching of relevant Categories by agent's Category

agent's Category	Relevant Category	Entities number
C1	C4, C5, C8, C9	90,000
C2	C6, C7, C8, C9	110,000
C3	C4, C5, C6, C7	70,000

7 Conclusion and Perspectives

In this paper, we proposed a solution to decrease processing time of a multiagent simulation without simplifying the context modeling. This issue is important because high execution times risk unfortunately to circumscribe the use of the multiagent paradigm to small-size simulations. Our proposal focused on the optimization of context computation. We propose to model a context as a filter, which allows us to propose a filters' structure as a tree and an algorithm for the agent to browse it efficiently. The proposed structure exploits the *a priori* cardinality of the different categories that an agent can take into account in the evaluation of his context. Our structure is simple and does not take into account the tests processed by the filters as an algorithm like RETE [8]. The advantages are a low memory cost and its independence against the environment dynamics. Our future work concerns the assessment of our proposal with distributed simulation like in [12] and the introduction of new data structures, such as lattices, in the organization of filters in order to take into account other filter's classification criteria. Our experimentation showed that our improvements became insignificant in comparison to the time to compute the set of perceptible data. Another perspective is to take into account the researches to optimize the environment data management like in [10,6]. In addition, we plan to enrich the evaluation of our proposal with several real world applications.

References

1. Badeig, F., Balbo, F.: Définition d'un cadre de conception et d'exécution pour la simulation multi-agent. Revue d'Intelligence Artificielle 26(3), 255–280 (2012)
2. Badeig, F., Balbo, F., Pinson, S.: Contextual activation for agent-based simulation. In: Proceedings of the 21st European Conference on Modelling and Simulation, ECMS (2007)
3. Béhé, F., Galland, S., Gaud, N., Nicolle, C., Koukam, A.: An ontology-based metamodel for multiagent-based simulations. Simulation Modelling Practice and Theory 40, 64–85 (2014)

4. Collier, N.: Repast: An extensible framework for agent simulation, vol. 36. The University of Chicago's Social Science Research (2003)
5. Abowd, G.D., Dey, A.K.: Towards a better understanding of context and context-awareness. In: Gellersen, H.-W. (ed.) HUC 1999. LNCS, vol. 1707, pp. 304–307. Springer, Heidelberg (1999)
6. Farenc, N., Boulic, R., Thalmann, D.: An informed environment dedicated to the simulation of virtual humans in urban context. In: Proceedings of EUROGRAPHICS 1999, pp. 309–318 (1999)
7. Ferber, J., Gutknecht, O.: Madkit: A generic multi-agent platform. In: 4th International Conference on Autonomous Agents, pp. 78–79 (2000)
8. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19, 17–37 (1982)
9. Kubera, Y., Mathieu, P., Picault, S.: Interaction-oriented agent simulations: From theory to implementation. In: Ghallab, M., Spyropoulos, C., Fakotakis, N., Avouris, N. (eds.) Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008), pp. 383–387. IOS Press (2008)
10. Michel, F.: Translating agent perception computations into environmental processes in multi-agent-based simulations: A means for integrating graphics processing unit programming within usual agent-based simulation platforms. Systems Research and Behavioral Science 30(6), 703–715 (2013)
11. Sierhuis, M., Clancey, W.J., Van Hoof, R.J.: Brahms: a multi-agent modelling environment for simulating work processes and practices. International Journal of Simulation and Process Modelling 3(3), 134–152 (2007)
12. Šišlák, D., Rehák, M., Pěchouček, M., Rollo, M., Pavlíček, D.: A-globe: Agent development platform with inaccessibility and mobility support. In: Software Agent-Based Applications, Platforms and Development Kits, pp. 21–46. Springer (2005)
13. Wagner, G.: AOR modelling and simulation: Towards a general architecture for agent-based discrete event simulation. In: Giorgini, P., Henderson-Sellers, B., Winikoff, M. (eds.) AOIS 2003. LNCS (LNAI), vol. 3030, pp. 174–188. Springer, Heidelberg (2004)
14. Warden, T., Porzel, R., Gehrke, J.D., Herzog, O., Langer, H., Malaka, R.: Towards ontology-based multiagent simulations: The plasma approach. In: 24th European Conference on Modelling and Simulation (ECMS 2010). European Council for Modelling and Simulation, pp. 50–56 (2010)