An algebra for modelling the simultaneity in agents' behavior in spatially explicit social-environmental models

Washington Sena de França e Silva[1], **Tiago Garcia de Senna Carneiro**[1]

[1]Departamento de Computação – Universidade Federal de Ouro Preto (UFOP)
Ouro Preto – MG – Brazil

wsenafranca@gmail.com, tiagogsc@gmail.com

Abstract. *Humanity is the major driver of spatial changes resulting from interactions between social and environmental systems. Environmental models usually apply the agent-based modeling paradigm to describe the social aspects of spatial changes. For this reason, these models have incorporated challenges inherent to this paradigm. One of these challenges is how to provide a semantically correct way to describe and simulate the simultaneity in execution of agents. In this context, this work describes an algebra to the development of spatially explicit agent-based models in a way that the algebra operators implicitly treat the simultaneity in agent's execution.*

1. Introduction

The most indicated modelling paradigm to describe social aspects of spatial change processes is the Agent-Based Model [Parker et al. 2003]. In this paradigm, different types of individuals (agents) are able to communicate and change the space through function that represents the agent's behavioral rules [Macal and North 2005]. However, this paradigm does not have a syntactic structure for directly representing heterogeneous spaces. This type of spatial structure needs different sets of attributes, resolutions and neighborhood relations at different locations. For this reason, it is useful to combine Cellular Automaton (CA) and Agent-Based Model (ABM) paradigms to describe biophysical and social aspects of spatially explicit environmental models [d'Aquino et al. 2002].

The use of the ABM presents issues that are inherent to this paradigm. Among these issues, there is a need to provide a semantically correct way to describe and simulate the simultaneity in execution of agents. This simultaneity means that agents may perform changes in the current state of simulation (present) considering the last synchronized state of simulation (past), and perceive the changes into environment only after they synchronize their own information [Michel et al. 2001]. Analogously, agents may perceive also the changes into environment instantaneously when they perform their changes from the present state into the same present state [Brown et al. 2005].

Ideally, the manner that agents perceive the environment or execute their behavioral rules should be independent of simulation platforms and their architectures (parallel or sequential). A same set of rules, i.e., a model should have the same semantics in any simulator. Despite this, platforms that use the sequential architecture tend to ignore the simultaneity in execution of agents. Consequently, these approaches deal with a strictly simultaneous behavioral rule as a sequential one. For this reason, simulations may present incorrect results caused by computational artifacts [Coakley et al. 2012].

On the other hand, platforms for simulation of agents that execute under some parallel architecture like REPAST-HPC [Collier and North 2011], FLAME [Coakley et al. 2012] and D-MASON [Cordasco et al. 2013], are able to deal with the simultaneity in the execution of agents. They deal with it using parallel programming and defining strategies to allow that tasks like scheduling, communication and synchronization [Shook et al. 2013, Fujimoto 2015, Rousset et al. 2016] perform in automatized and transparent way from the modeler's perspective. This way, modelers can explore high performance simulations even when they are inexperienced to deal with parallel programming issues. However, these approaches force modelers to describe behavioral rules in a manner that such rules have to guarantee the coherence and consistency of simulations. This can be highlighted in cases in which exist concurrent access to shared resource like in collision avoidance [Torrens and McDaniel 2013], matching and reproduce [Lysenko et al. 2008].

An alternative way to guarantee coherence and consistency in simulations is to provide such simultaneity control in the language level instead of doing it in level of the simulation engine. In this way, modelers can clearly express the expected semantics from his/her code diminishing ambiguity in rules semantics. Sequential architectures will find in the model code the information required to simulate simultaneity. Parallel architectures will find the necessary information to simulate sequential behaviors. It is possible to guarantees that a correct model will perform a correct simulation as well. In this context, this paper defines and evaluates one approach for the specification of simultaneity in the execution of agents through an algebra for spatially explicit agent-based model development.

This paper is organized as follows. Section 2 highlights some related works. Next, section 3 describes the algebra, its types, operators, syntax and semantics. Section 4 shows the experiments developed to demonstrate how the algebra has solved some problems faced in agent's modelling and simulations and then we present the algebra usage through a classical model. Finally, section 5 presents the conclusions of this work.

2. Related Works

Providing the simultaneity in execution of agents in modelling level is a manner to guarantee consistence and coherence for semantically correct models in simulation time. In these approaches, the modelling language is able to deal with these issues. The early works to present solutions for this are DESIRE [Dunin-Keplicz and Treur 1994], Concurrent METATEM [Fisher 1994], ConLog [De Giacomo et al. 2000] e AALAADIN [Michel et al. 2001].

Recently, the works that much relate to our approach are the languages ALOO and SARL. The agent-oriented language ALOO [Ricci and Santi 2013] uses the concept of agents (mobility entities) and objects (stationary entities) to define one semantic for mutual exclusion on language level for scenarios where several agents are trying to access or change a same object. ALOO is therefore able to guarantee concurrency control in accesses to shared resource.

The general-purpose agent-oriented programming language SARL [Rodriguez et al. 2014] provides a manner to encapsulate the model's partition, and the communication and synchronization of agents through concepts of multi-contexts and spatial hierarchy. Briefly, agents can communicate only with other agents that are located at same space and are able to access the same context. In this way, the language make explicit the groups of agents that are able to communicate and therefore, need to keep their states synchronized.

Comparing the previous approaches with our own. Both ALOO and our approach have mechanisms to guarantee coherence in a scenario where exist concurrent access to shared resource in a way that modelers do not need to deal with the concurrency control directly. Comparing our algebra and SARL, they both use concepts as groups of agents in language level for providing coherency in communication and synchronization of agents.

The main aspects that differ our algebra and these languages are: (1) Modelers can clearly express the expected semantics for agent's rules; (2) The algebra provides two ways (simultaneous or sequential) to execute a same agent's rule.

3. An Algebra for describing social-environmental spatially explicit model

An algebra specifies his components in a manner that makes possible to abstract the implementation of these components [Frank 1999]. Hence, algebras for ABM are independent of programming languages and of simulator's architectures as well. In this paper, we define an algebra by a set of types and operators applicable to these types.

3.1. Types

Types in an algebra are the kind of entities that the algebra's operators are able to manipulate. Types define in which kind of entities the modeled phenomenon can be decomposed and represented. In this work, modelers are able to describe their models in terms of agents, collections of agent, cellular spaces, and social and spatial relations. These types are grouped into three main categories: (1) basic, (2) collections and (3) relations.

An agent is a basic type that performs changes in the environment. An agent has an attribute list. Each attribute in this list is a pair key-value (Figure 1). A key represents the name of an agent property and the value represents the current state of the correspondent property.

$Attribute : (Key, Value)$
- $Key : Indentifier$
- $Value : Boolean|Number|String|Agent|Cell|[Attribute]|Null$

Figure 1. Formal definition of an attribute.

All agents have a non-null attribute that locate them in the space. This attribute is a reference to a certain cell. The main function of this attribute is to enable agent movement. To perform this movement, an agent just need to replace the current value of his location attribute by another cell. A cell describes properties of a spatial location. Besides the attribute list, a cell also has an agent list to store all agents placed inside it and a list of its neighbor cells (Figure 2a).

A collection represents a set of same-type entities. The figure (Figure 2b) shows the definition of all collections in this algebra. A society is a collection for same-type agents. Two agents have the same type when they present the same internal states and the

(a)

$Agent : (Attributes, Location)$
- $Attributes : [Attribute]$
- $Location : Cell$

$Cell : (Attributes, Agents, Neighbor)$
- $Attributes : [Attribute]$
- $Agents : [Agent]$
- $Neighbor : [Cell]$

(b)

$Society : Agents$
- $Agents : [Agent]$

$Group : Agents$
- $Agents : Sorted([Agent])$

$CellularSpace : Cells$
- $Cells : [Cell]$

Figure 2. (a) Definition of basic types. (b) Definition of agent and cell collections.

same behavioral rules. Group is a set of agents in which every agent must satisfy a given selection function. For example, a group of agents of same gender or a group of agents where every agent is older than a given age. A group can sort its agents to define a kind of precedence between them. Thus, groups are filters defined over societies, selecting the agents that will activated in some action.

A Cellular Space is a grid of cells described by the same attributes. Each cell has a set of cells that defines its neighborhood. This neighborhood is essential to simulate spatial process using the CA paradigm.

A relational type is responsible for connecting agents enabling communication between them. The relational type social network can represents any relation between agents. A modeler defined function generate a social network. This function must determine the weight of a connection between two agents in a society. A weight with 0% means that there is no connection and 100% means a connection of maximum intensity. In a social network, agents are nodes in a graph, their connection are edges and the edges' weight are the strength agent's connection [Andrade et al. 2010]. In this algebra, social networks are like maps in which agents work as indexes which maps to lists containing agents and weights representing their connections (Figure 3).

$SocialNetwork : Agent : Connections$
- $Connections : [(Agent, Weight)]$
- $Weight : Number$

$SpatialNeighborhood : (CellularSpace, N, M)$
- $N : Number$
- $M : Number$

Figure 3. Definition of algebra's relational types.

Besides the social network, neighborhoods represent spatial relations between agents. In this relation, an agent connects to another agent through the cell's neighborhood structure. Using the cell's list of agents, an agent can access all agents from a neighbor cell. In this manner, agents are able to connect by proximity relations. A spatial neighborhood is defined by a reference to a cellular space and by its dimension. The size of a neighborhood is a pair N and M (MxN), where N is the number of cells in vertical and M is the number of cells in horizontal.

3.2. Operators

Operators are a set of functions applicable to the types previously presented. Next, we present the syntax and semantic of each operator defined in this algebra.

The ask operator uses a message passing schema for providing interaction between agents and other types (Figure 4). Agents send messages to receivers requesting them to perform some actions (tasks).

$ask : Function(Receiver, Action, Args)$
- $Receiver : Agent|Cell|Society|Group|CellularSpace|[Receiver]$
- $Action : Function(Receiver, Args)$
- $Args : [*]$

Figure 4. Definition of ask operator.

Since actions like move, die and reproduce are very common in ABM, these operators were pre-defined in this algebra using the ask operator (Algorithm 1).

Algorithm 1 Defining operators move, die and reproduce using operator ask.

```
function MOVE($ag$, $cell$)
    $loc = ag.location$
    $ask(loc.agents, pop, ag)$
    $ask(ag, setLocation, cell)$
    $ask(cell.agents, push, ag)$

function DIE($ag$)
    $loc = ag.location$
    $ags = loc.agents$
    $ask(ags, pop, ag)$
    $ask(soc, pop, ag)$

function REPRODUCE($ag$)
    $loc = ag.location$
    $child = copy(ag)$
    $ask(society, push, child)$
    $move(child, ag.location)$
```

In this algebra, there is not a way for directly create basic types. Modelers must use collection construction operators to instantiating entities of these types. In this manner, every basic entity will be enclosed in at least one collection. Figure (Figure 5) briefly defines the construction operators for collections and relations.

$createSociety : Function(Instace, Quantity) \rightarrow Society$
- $Instace : Agent$
- $Quantity : Number$

$createGroup : Function(Society, Filter, Compare) \rightarrow Group$
- $Filter : Function(Agent) \rightarrow Boolean$
- $Compare : Function(Agent, Agent) \rightarrow Boolean$

$createCellularSpace : Function(Instace, Dimension) \rightarrow CellularSpace$
- $Instace : Cell$
- $Dimension : (Width : Number, Heigh : Number)$

$createSocialNetwork : Function(Society, Connection) \rightarrow SocialNetwork$
- $Connection : Function(Agent, Agent) \rightarrow Number$

$createSpatialNeighborhood : Function(Space, N, M) \rightarrow SpatialNeighborhood$
- $Space : CellularSpace$
- $N : Number$
- $M : Number$

Figure 5. Definition of agent and cell collections.

The construction operator of society creates a society using an agent definition and a given quantity. This definition works as a template that enables the operator to instantiate any quantity of agents in a society. The operator creates each agent as a copy of the archetype agent. The parameter quantity determines the number of agents that the operator will create. The construction operator for group uses a society, a selection function and a compare function to create a group. In the same way, the construction operator for cellular spaces instantiates cells by copying the archetype cell received as parameter. The cellular space dimension determines the quantity of cells that the operator will create. The construction operator for social network uses a society and a function that determines the intensity of connections between each pair of agents to create a social network. The construction operator for spatial neighborhood uses a cellular space and the required neighborhood dimension to create a spatial neighborhood.

Modelers should use execution operator to simulate collection of agents provoking changes described by the behavioral rule received as parameter. The modeler defines these rules as functions that govern the behavior of some types of agents. This approach allows the reuse of rule definitions, allowing the modeler to apply them to any collections able to execute it. Three factors determine the semantics of the operator execute: The type of collection received as parameter, the use of any relational as parameter, and the type of behavioral rule received as parameter (Table 1).

Table 1. Syntactic and Semantic definition of execute operator.

Operator	Syntax	Semantic
Simultaneous local execution	$execute(Society, Rule)$ • $Rule : Function(Agent)$	Each agent in a given society simultaneously applies a given rule independently of the other agents.
Sequential local execution	$execute(Group, Rule)$ • $Rule : Function(Agent)$	Each agent in a given group sequentially applies a given rule independently of the other agents.
Simultaneous shared execution	$execute(Society, Relation, Rule)$ • $Relation :$ – $SocialNetwork$ – $SpatialNeighborhood$ • $Rule : Function(Agent, Agent)$	Each agent in a given society simultaneously applies a given rule that enables communication between agents.
Sequential shared execution	$execute(Group, Relation, Rule)$ • $Relation :$ – $SocialNetwork$ – $SpatialNeighborhood$ • $Rule : Function(Agent, Agent)$	Each agent in a given group sequentially applies a given rule that enables communication between agents.

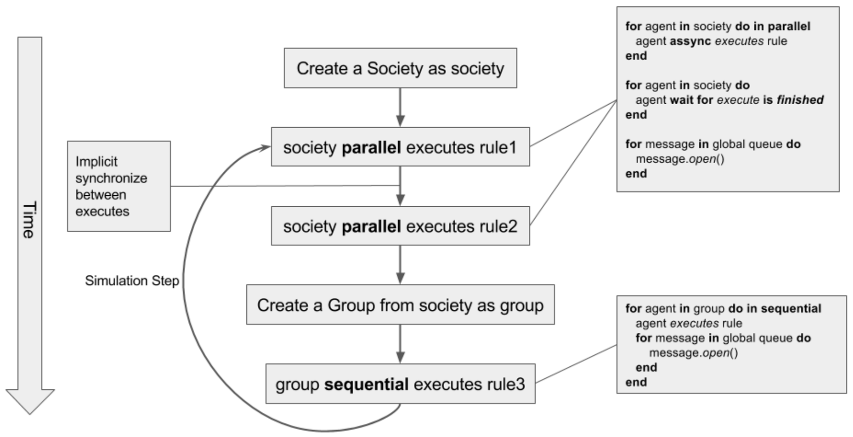
When a society executes a rule, all agent simultaneously performs changes. This means that agents will perceive the provoked changes only after all of them have accomplished their execution. A group enables that agents instantaneously perceives any provoked change. Group executes agent by agent in a sorted and sequential manner. This mode of execution guarantees mutual exclusion for agents performing the same rule. In this context, the rule code works as a critical section. Thus, agents perceive changes as soon as each agent finishes its execution. The group's order function determines in which order agents will execute.

Relational types determines how the communications between agents of a given collection will occur. When execute operator does not receive a relation, changes are local. This mean that, an agent will apply a change independently of the others. When an execution has a relation, the rules will receive two agents as parameter. The first agent will apply the rule while the second one will only take in a communication process. Usually, these rules describe behaviors that collect information from other agents to support the decision-making process. The second agent is a read-only object. The only way to change the value of an attribute from the second agent is through requests using the *ask* operator. This is a convention in order to guarantees coherent computations for all agents. Requests sent through the *ask* operator will be served only after the simulation synchronization stage, causing the changes requested.

The execute operator is also responsible for performing communication and synchronization of agents. Synchronization of agents is transparent to the modelers. Modelers do not need to deal with concurrency control to guarantee coherent computation. For this, the ask operator sends asynchronous messages and senders do not need to wait for receivers' responses to resume their execution. The execute operator will synchronize agents and process messages according to the semantics desired by the modeler, depending on the type of collection received as parameter: Society or group.

When a society invokes the execute operator, all agents perform their simulations in parallel and a synchronization barrier forces agents to wait until all other agents to finish. Only then, all agents will process the received asynchronous messages. This guarantees that all agents' rules will execute taking in consideration the same model state, which immediately precedes the invocation of the execute operator. In addition, no communication happens while agents' internal states are changing.

On the other hand, when a group invokes the execute operator, each agent will execute the behavioral rule sequentially. Immediately after the execution of each agent, all agents will perform the received messages and all agents will perceive the changes caused by the last behavioral rule executes (Figure 6).



Time
Create a Society as society
society **parallel** executes rule1
society **parallel** executes rule2
Create a Group from society as group
group **sequential** executes rule3
Implicit synchronize between executes
Simulation Step

```
for agent in society do in parallel
  agent assync executes rule
end

for agent in society do
  agent wait for execute is finished
end

for message in global queue do
  message.open()
end
```

```
for agent in group do in sequential
  agent executes rule
  for message in global queue do
    message.open()
  end
end
```

Figure 6. Conceptual model of algebra's execution.

In this semantic, if a rule demands an intermediate synchronization, the modeler should split the rule's code into two rules: (1) One containing the code that precedes the synchronization point, and (2) another one containing the code that comes after the synchronization point.

To execute more than one society at the same time, the modeler should use a list of tuples (Figure 7). When an execution will perform a local rule, a tuple has a society and a local rule. In case of a rule demands communication between agents, the tuple will contain a relational type as well. Semantically, a tuple execution is equals to the execute operator (Table 1). In a practical manner, all tuples simultaneously execute before that agents perceive any change (communication and synchronization). Execution by tuples allows for example that two different societies, using different rules, change the space at the same time.

$ExecutionTuple : (Society, Rule)|(Society, Relation, Rule)$
- $Relation : SocialNetwork|SpatialNeighborhood$
- $Rule : Function(Agent)|Function(Agent, Agent)$

Figure 7. Definition of an execution tuple.

4. Experiments

We have done two kind of experiments. (1) The first one demonstrated how the Algebra solves some problems that are related with the simultaneity and how they may affect the simulation results. In addition, these experiments also demonstrated how the simultaneity and the semantic of execution are related. (2) The second one demonstrates the algebra usage in implementing of Predator-Prey model. This classical model has features that are relevant and frequently used in spatial-explicit social-environmental models.

4.1. Effects of simultaneity in simulation results

The experiments demonstrated the effects of simultaneity in the simulation results using three simple models (Figure 8). In the first model, agents are trying to change simultaneously the energy in one shared cell of space (Figure 8a). When simulated through a society, changes performed by most of these agents have no effects. Agents update the cell computing their rules from the simulation past state, overwriting changes in the current simulation state and ignoring any other changes previously simulated. This way, only changes performed by one unique agent will be persisted and perceived in the future computations. On the other hand, when simulate via groups, changes are sequenced and agents will perceive the changes instantaneously.

In another model, eight agents are simultaneously trying to move to one a same empty cell (Figure 8b). Disregarding the collision, simultaneity semantics have shown to have a huge impact in model results. The execution by society resulted in a scenario where all agents moved to the same cell. In this case, they sensed the environment's state where this cell was empty. In the other hand, execution by group resulted in another mobility pattern, in this case only one agent has moved to the target cell. This because after his move the other agents perceived the environment's state where this cell was not empty anymore.

The third model describes a rule where each agent have to collect information from neighbor cells to decide whether he should put fire in his location (Figure 8c). When simulating this model through groups, the order of execution of the agents affects the simulation results, possibly introducing computational artifacts in it. However, this artifact did not appear when agents' rules are simulated through a society, because agents sense and change the space simultaneously, there is order in agents' execution.

	(a) Model	(b) Model	(c) Model

	`function change(ag, cell)`
	`    ask(cell, addEnergy, 2)`
	`end`
		`function moving(ag, cell)`
		`    if cell is empty then`
		`        move(ag, cell)`
		`    end`
		`end`
			`function burn(ag, cell)`
			`    c=count(cell.neighs,'burning')`
			`    if c >= 2 then`
			`        ask(cell, burn, true)`
			`    end`
			`end`

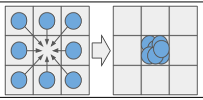
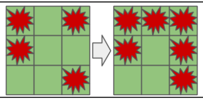
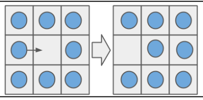
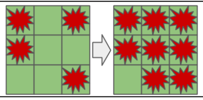
Society	cell.energy: 2		
Group	cell.energy: 4		

Figure 8. Comparison between the simulation using society and group

These experiments shows that a unique model code (syntax) can lead to different results depending on the simultaneity semantics adopted by a given modeling language or simulation engine. In the proposed algebra, modelers can clearly state the expected semantic diminishing ambiguity in model specification and allowing for simulations that produce exactly the same result no matters whether models are been executed by sequential simulators or not. Modelers may choose which semantics fits better the phenomenon being modeled. In addition, the algebra promotes code reuse allowing modelers to apply a same rule with different semantics.

4.2. Description of Predator-Prey model in the proposed algebra

The Predator-Prey model used in these experiments is an adaptation of Wilensky's version of Wolf-Sheep [Wilensky 1997]. In this simplified version, there are three types of entities: Wolves, Sheep and Space. Wolves and Sheep are agents that, in each simulation step, randomly move in space. Agents spend energy to move and dies once they spent all energy. Agents eat to recover their energy. Sheep eat grass from their cells. Wolves prey sheep that are in their cell. The space is modeled as a cellular automaton that simulates grass periodical regrowth. Agents also reproduce by losing half of their energy for the newborn agents.

This model demonstrates the usage of execute and ask operators in describing interactions between two different societies, which individuals compete for access to shared resources (preys and grass). The algorithm (Algorithm 2) shows the *hunting* rule of predators. If a predator meets a prey, predator will target this prey. After marking a prey as target, a predator will attack it. The algorithm (Algorithm 2) describes the behavior of an attack. The *attacking* rule recovers predator's energy by the amount of target's energy and informs that the target is going to lose its energy and then die after the next synchronization.

One can interpret the prey's behavior analogously to predator's behavior (Algorithm 2). Preys will target their own location cells. Then, preys will eat grass from these cells. This case shows that different types of agent can perform a same rule since theys

Algorithm 2 Behavioral rules of hunting.

```
function HUNTING($predator, prey$)
    $predator.target = prey$
function TRYEAT($prey$)
    $prey.target = prey.location$
function ATTACKING($predator$)
    $predator.energy += predator.target.energy/2$
    $ask(predator.target, setEnergy, 0)$
```

have a same set of internal states. In this experiment, preys and predators have energy and target as attributes, and cells have the unique attribute energy (Algorithm 3). In this manner, any agent can perform the *attacking* rule.

Algorithm 3 Defining predator, prey and cell.

```
$predator = Agent\{energy = 40, target = null\}$
$prey = Agent\{energy = 40, target = null\}$
$cell = Cell\{energy = 40\}$
```

The algorithm (Algorithm 4) shows the execution of predator and prey rules. Predators and preys must sense the same environment state. Therefore, they must execute simultaneously. For this, agents execute their rules (moving, hunting and tryEat) using two tuples (*movingExecuteTuples* and *huntingExecuteTuples*). These tuples allow both societies to execute at same time.

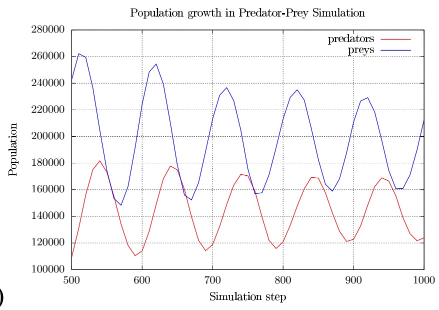
In contrast, some agent behavior demands that only one agent executes per time in order to guarantee coherence to model results. For instance, only one predator can kill a given prey. Therefore, a group must execute the *attacking* rule sequentially, meaning that only one agent will attack a target. This way, the *attacking* rule is a critical section of code in which mutual exclusion to resources (target) is guaranteed and all agents will sense attacks at the same instant as they occurs.

Algorithm 4 Scheduling for executions of predators and preys.

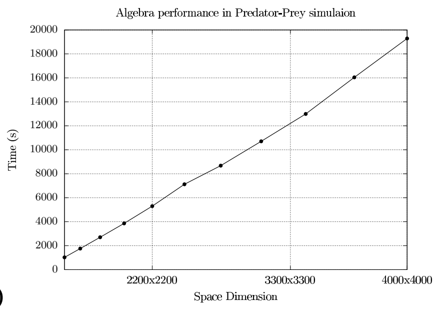
```
function MAIN($POPULATION, DIMENSION$)
    $predators = createSociety(predator, POPULATION)$
    $preys = createSociety(prey, POPULATION)$
    $space = createCellularSpace(cell, DIMENSION)$
    $neighs = createNeighborhood(space, 1, 1)$
    $movingTuples = [(predators, moving), (preys, moving)]$
    $huntingTuples = [(predators, neighs, hunting), (preys, tryEat)]$
    for $t = 1...1000$ do
        $execute(movingExecuteTuples)$
        $execute(huntingExecuteTuples)$
        $predatorsGroup = createGroup(predators, hasTarget)$
        $preysGroup = createGroup(preys, hasTarget)$
        $execute(predatorsGroup, attacking)$
        $execute(preysGroup, attacking)$
```

Group also filters the society allowing only few agents to execute the attacking rule, the ones who have targets. The *predatorsGroup*) and preys (*preysGroup* groups do not have an order function defined by the modeler. By default, groups will randomly organize their agents. Hence, all agents have the same chance to attempt an attack to a prey.

In order to evaluate the performance that can be attained by a C++ and OpenMP [Dagum and Menon 1998] implementation of this algebra (Figure 9), the predator-prey model was simulated for spaces of different sizes and, therefore, different population sizes.



(a)



(b)

Figure 9. (a) Population growth of predators and preys in a cellular space of dimension 1000 x 1000. (b) Performance results of Predator-Prey simulation.

Initial experiments have shown that simulations using the proposed algebra may achieve a performance curve near to linear in relation of the number of cells in space. This demonstrates that this algebra is also a viable solution from the performance point of view.

5. Conclusions

This paper has presented an algebra for modeling the social aspects of spatial changes in accordance to the Agent-Based Model paradigm. Experiments have demonstrated how this algebra can handle some problems that relates to the simultaneity in execution of agents. Beside this, experiments have demonstrated also the usage of the algebra in the development of a model that has features that are common to many spatially explicit socio-environmental models. The contributions of this algebra are as follow:

1. To allow the definition of behavioral rules independently of the agents that will execute them.
2. To show one way of decoupling model description from the issues that rises from the parallel simulation of multiple agents.
3. To allow for modeler decides the execution semantics of agent's rules.

For these reasons, we believe that this algebra can facilitate the development of models that use the agent-based modeling paradigm. It is still necessary to evaluate the algebra in the development and simulation of other models. Thus, determining if there are models that this algebra is not sufficient to describe them. Furthermore, we wish to evaluate this algebra in large-scale simulations in order to understand the pros and cons of this approach from the high performance point of view.

References

Andrade, P. R., Monteiro, A. M. V., and Camara, G. (2010). Entities and relations for agent-based modelling of complex spatial systems. In *Social Simulation (BWSS), 2010 Second Brazilian Workshop on*, pages 111–118. IEEE.

Brown, D. G., Riolo, R., Robinson, D. T., North, M., and Rand, W. (2005). Spatial process and data models: Toward integration of agent-based models and gis. *Journal of Geographical Systems*, 7(1):25–47.

Coakley, S., Gheorghe, M., Holcombe, M., Chin, S., Worth, D., and Greenough, C. (2012). Exploitation of high performance computing in the flame agent-based simulation framework. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 538–545. IEEE.

Collier, N. and North, M. (2011). *Repast HPC: A platform for large-scale agentbased modeling*. Wiley.

Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., and Spagnuolo, C. (2013). Bringing together efficiency and effectiveness in distributed simulations: the experience with d-mason. *Simulation*, 89(10):1236–1253.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

De Giacomo, G., Lespérance, Y., and Levesque, H. J. (2000). Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1):109–169.

Dunin-Keplicz, B. and Treur, J. (1994). Compositional formal specification of multi-agent systems. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 102–117. Springer.

d'Aquino, P., August, P., Balmann, A., Berger, T., Bousquet, F., Brondízio, E., Brown, D. G., Couclelis, H., Deadman, P., Goodchild, M. F., et al. (2002). Agent-based models of land-use and land-cover change. In *Proc. of an International Workshop*, pages 4–7.

Fisher, M. (1994). Representing and executing agent-based systems. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 307–323. Springer.

Frank, A. U. (1999). One step up the abstraction ladder: Combining algebras-from functional pieces to a whole. In *International Conference on Spatial Information Theory*, pages 95–107. Springer.

Fujimoto, R. (2015). Parallel and distributed simulation. In *Proceedings of the 2015 Winter Simulation Conference*, pages 45–59. IEEE Press.

Lysenko, M., D'Souza, R. M., et al. (2008). A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4):10.

Macal, C. M. and North, M. J. (2005). Tutorial on agent-based modeling and simulation. In *Proceedings of the 37th conference on Winter simulation*, pages 2–15. Winter Simulation Conference.

Michel, F., Ferber, J., and Gutknecht, O. (2001). Generic simulation tools based on mas organization. In *10th European Workshop on Modelling Autonomous Agents in a Multi Agent World MAMAAW*, volume 1.

Parker, D. C., Manson, S. M., Janssen, M. A., Hoffmann, M. J., and Deadman, P. (2003). Multi-agent systems for the simulation of land-use and land-cover change: a review. *Annals of the association of American Geographers*, 93(2):314–337.

Ricci, A. and Santi, A. (2013). Concurrent object-oriented programming with agent-oriented abstractions: the aloo approach. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*, pages 127–138. ACM.

Rodriguez, S., Gaud, N., and Galland, S. (2014). Sarl: a general-purpose agent-oriented programming language. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on*, volume 3, pages 103–110. IEEE.

Rousset, A., Herrmann, B., Lang, C., and Philippe, L. (2016). A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review*.

Shook, E., Wang, S., and Tang, W. (2013). A communication-aware framework for parallel spatially explicit agent-based models. *International Journal of Geographical Information Science*, 27(11):2160–2181.

Torrens, P. M. and McDaniel, A. W. (2013). Modeling geographic behavior in riotous crowds. *Annals of the Association of American Geographers*, 103(1):20–46.

Wilensky, U. (1997). Netlogo wolf sheep predation model. *URL http://ccl. northwestern. edu/netlogo/models/WolfSheepPredation. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.*