Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

EXTENDED MODEL SPACE SPECIFICATION FOR MOBILE AGENT-BASED SYSTEMS TO SUPPORT AUTOMATED DISCOVERY OF SIMULATION MODELS

Hai Le

Department of Computer Science
Georgia State University
25 Park Pl NE
Atlanta, GA 30303, USA

Xiaolin Hu

Department of Computer Science
Georgia State University
25 Park Pl NE
Atlanta, GA 30303, USA

ABSTRACT

Automated discovery of simulation models is a different simulation modeling approach from the traditional approach, where simulation models are handcrafted by modelers. Our previous work developed an automated simulation modeling approach for mobile agent-based systems that allows automated search of candidate models based on a search space and desired simulation behaviors. This paper extends the model space specification from previous work to support an expanded search space for automated discovery of simulation models. The extended specification includes supporting user-defined properties to capture internal states and other properties of mobile agents and adding a new Activation component for behaviors so that priorities among multiple behaviors can be dynamically computed based on surrounding environment. The extended specification is demonstrated by supporting discovery of simulation models that are not in the previous search space.

1 INTRODUCTION

Developing simulation models for complex systems is a challenging task. Traditionally, modelers use their knowledge or consult experts to create a set of initial models that capture the behavior and structure of a system under study. Then, the initial models are gradually improved until an end result meets the simulation requirements. This approach is beneficial when testing theories of how a system works. However, the handcrafted models often have biases from their creators. Furthermore, it takes increasing time and effort to develop high quality models as the system complexity increases. To alleviate this problem, in previous work we developed a new simulation modeling approach that aims to support automated model discovery for mobile agent-based systems (Keller and Hu 2019). The main feature of the approach is to define a model space representing possible models, and a search method (Genetic Algorithm) to find candidate models based on specified criteria. The approach has been shown to be able to discover a variety of interesting models for mobile agent-based systems.

A key component of the developed modeling approach is the model space for searching candidate models. To define an effective model search space, a formal model specification is important so that automated model discovery is possible. Our previous work provided a basic model specification for mobile agent-based systems where a world includes a 2D space and a set of agents. Each agent has several pre-defined properties including position, (moving) direction, speed, and one or more behavior groups that manipulate properties' values. At each time step, agents execute their behaviors in behavior groups to sense the surrounding environment and move accordingly in the 2D space.

While the previous model specification was shown to work well for a variety of models, it has two major limitations. First, it works only with a set of predefined agent properties. These predefined properties are position, direction, and speed, which are basic to mobile agent-based systems. These basic properties are sufficient to model relatively simple scenarios. For more complex scenarios, agents need to have more complex properties in order to capture their internal states or other properties. For example, an agent may need an energy property to reflect how much remaining energy the agent has. The energy can increase or decrease based on what the agent chooses to do: it decreases if the agent moves and increases if the agent recharges itself. To allow adding new properties such as the energy property into the model space for automated model discovery, the model specification needs to be extended to include a formal specification for agent properties so that new properties can be added in a well-defined manner. Because the specific properties to be added are open-ended, the agent property specification needs to be genetic in order to cover a wide range of potential properties for different applications.

The second limitation is that the moving actions of all behaviors are averaged without a mechanism to add priorities among the different behaviors. The approach of averaging moving actions from multiple behaviors is not uncommon in the literature. For example, in the boids model (Reynolds 1987) an agent's steering direction in each step is averaged from the separation, alignment, and cohesion behaviors. However, there are many other situations where prioritizing behaviors would work better when computing an agent's final movement. Consider the behavior of obstacle avoidance, which moves the agent away from the obstacle, as an example. When the agent is relatively far away from the obstacle, it is less important to incorporate the moving action of this behavior into the overall movement. As the agent moves closer to the obstacle, this behavior becomes more important: the closer the agent is from the obstacle, the more important the behavior is. In other words, the moving action of this behavior becomes more dominant as the agent gets closer to the obstacle. To support this capability, there is a need to prioritize the behaviors based on how "important" the behaviors are.

To address the two limitations discussed above, this paper extends the model specification in previous work to support automated discovery of simulation models for mobile agent-based systems. We add two major extensions on top of previous work. First, we provide a formal and generic specification for agent properties so that new properties can be specified in a uniform and well-defined way. This allows modelers to add customized new properties to a model space when searching for candidate models for specific applications. Second, we extend the behavior specification of agents so that each behavior has two components: an Activation component and an Action component. The Activation component specifies the level of activation of the behavior. This allows the priority of a behavior to be modeled because different activation levels represent different priorities. The Action component specifies the action of the behavior, i.e., how the behavior changes the value of a property. This component is similar to what we had in previous work. Both the Activation component and Action component dynamically compute their outputs based on agents' conditions and surrounding environment. Differentiating these two components makes it possible to define behaviors that are dynamically activated based on conditions of the environment and then act accordingly to respond to the environment. This in turn supports discovery of more complex models and increases the applicability of the modeling approach to more applications. In this paper, we present the extended model specification and describe how the overall framework works, and show several examples of automated model discovery based on the extended model specification

2 RELATED WORK

The challenge of agent-based modeling and simulation is discovering agent behaviors so that they all can simulate desired scenarios. Traditional approach using domain knowledge has been used to build models that are resemble desired scenarios. For example, in Reynolds' famous boids model (Reynolds 1987) resembles behaviors of flock of birds. In (Deeter et al. 2004), the researchers assign trigger, script, and task roles for agents to resemble different components of an intrusion detection system. Another example is (Wagner and Agrawal 2014) where a prototype is built to study crowd behaviors when there is the presence of a fire disaster. A disadvantage of this approach is model development are limited by knowledge of creators. Hence, handcrafted models have bias from them, and in many cases, the results do not reflect every aspect of the systems. However, handcrafted models are usually understandable because they are delivered from human knowledge.

Instead of relying on domain knowledge, several data-driven approaches are used to discovery of simulation model automatically. The works in (Lee et al. 2007) and (Gianluca et al. 2004) extract ground truth data from video clips of human movements, then the researchers learn crowd behavior model from observed trajectories. If given enough data, this technique can resemble how crowds behave in real life scenarios. However, it does not provide explanation behind the discovered behaviors. Furthermore, they are only work with particular application where historical data was collected. Other works uses data farming technique (Horne and Meyer 2004) to generate a wide range of models from a set of possible parameters. Many works in this approach include (Huang and Verbraeck 2009) and (Beck et al. 2015) concentrate on adjusting parameters to make well-calibrated models that reproduce the real system, but not focus on the improvement of model specification.

In this work, we add two major components to our framework to support more complex and open-ended applications. First, (Decraene et al. 2010) suggests that to make evolutionary computation possible, parameters must have type (*numerical, enumerable*) and range (min, max). Base on this, we provide a formal and generic specification for agent properties so that new properties can be specified in a uniform and well-defined way. Second, (Kendall et al. 2000) and (Bhouri et al. 2012) describe the importance of priority between agents on mobile agent-based system. In our work, instead of focusing on agent priority, we add a mechanism to assign priority to each agents' behavior. Hence, this make the framework assist better when agents decide overall movement action from multiple behaviors.

3 OVERVIEW OF PREVIOUS WORK

Our previous work developed a framework towards data-driven simulation modeling for mobile agent-based systems (Keller and Hu 2019). The framework includes two major components: a model space specification and a search algorithm.

The model space specification provides a formal specification for the general model structure from which various models can be generated. It defines a "meta-model" for behaviors-based mobile agent systems. A mobile agent-based system is composed of a world and a set of agents. A world is a 2-D space that wraps vertically and horizontally. All agents have the same set of behavior groups. Each behavior group contains a set of behaviors, and all behaviors in a behavior group manipulate the same pre-defined property such as: position, heading direction, or speed. Properties play critical roles because each behavior acts on a specific property by changing its value based on observations of the environment and nearby agents' properties. Detailed specification of the model space can be found in (Keller and Hu 2019).

Based on the model space specification, in each iteration of the discrete time simulation, a behavior goes through multiple steps to compute a moving action of the behavior. Figure 1 illustrates these steps where each step is represented by a box. The arrows before and after the boxes represent the input and output of these steps. In the first step, agents sense a set of nearby neighbors within their field of view (FOV). The second step applies filters to get a specific set of agents. The next step extracts property from the set of agents and combines them into one reference value. Then, the last step is to add an offset to the reference value to compute a desired value for the property the behavior is acting on. This desired value is referred to as the desired action for this behavior. If there are multiple behaviors that belong to the same behavior group, the described actions are averaged to compute an overall action of the agent. Note that a special property is the position property, which is treated differently based on which step a position value is checked. Specifically, in the *extract action property* step, a *position* value is always converted to the direction towards that position; the direction then becomes the reference value for the next step (the *add action offset* step). In all other situations (including the steps in the Activation components described in Section 4.2.1, a position value is treated as a value for calculating a distance to that position.

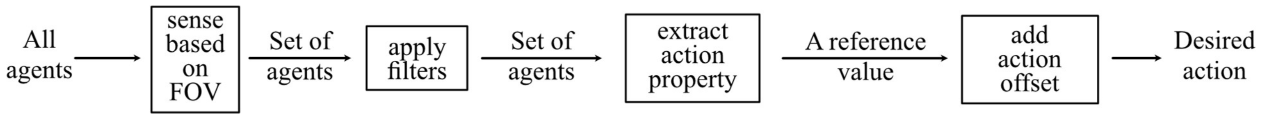


Figure 1: Steps of a behavior.

Figure 2 illustrates the behavior steps for a sample behavior that makes an agent steer left from its nearest neighbor. In this example, the 2D space includes ten regular agents (hollow white circle) and one obstacle (black solid circle), where agent A is the agent that owns this behavior. Through step 1, there are five entities: four regular agents (1-4) and one obstacle within agent A's FOV. In step 2, agent A applies couple of filters to eliminate unsatisfied neighbors. The first filter chooses regular agent only (step 2.a). Hence, the obstacle is removed. The second filter chooses the nearest neighbor (step 2.b), and as a result agent 1 is selected. In step 3, agent 1's position property is extracted and then converted to the direction towards that position. This direction becomes the reference value for the moving action of this behavior. Then step 4 adds an offset value to the reference direction to compute the desired action of this behavior. In the last step, agent A executes the desired action and moves according to the new direction.

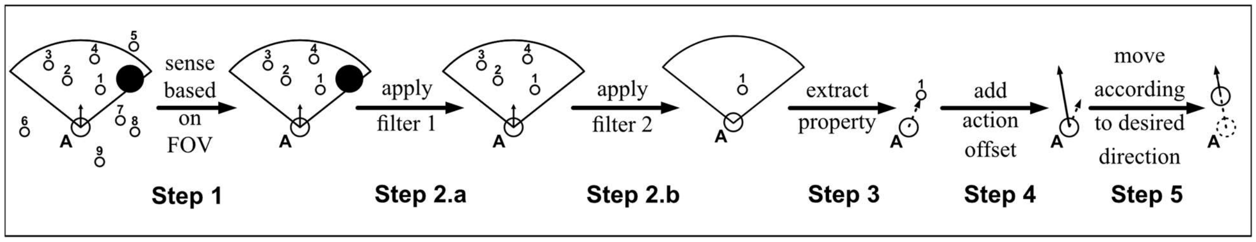


Figure 2: Illustration of the behavior steps.

Based on the model space, Genetic Algorithm (GA) is used to find models that satisfy the desired behavior patterns specified by a modeler. GA encodes a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination on these structures so as to preserve critical information (Whitley 1994). Based on the specification, a random model can have one or more behavior groups. Each behaviors group contains one or more behaviors. Each behavior has several choices to choose for range filters (distance, angle, speed, etc.), method filters (nearest, furthest, average, etc.), and extract property functions (position, direction, speed). These options are considered as chromosomes for models, and all possible combinations between them create the search space. First, random models are generated within the search space, and all of them create an initial population. Next, simulation of each model is performed by using a set of agents that make decisions based on model's behavior group specifications. A set of fitness functions is used to evaluate how close the outcomes of models are to the simulation goals. After the evaluation, if the fitness scores of one or more models pass the threshold, GA finds the best models and stops. If they do not pass, a new population is created with 25% best from old generation, 25% mutation from 25% best of old generation, and 25% crossover from the all population. The last 25% is randomly generated to increase the diversity of the populations (Keller and Hu 2019), and a new circle begins until GA finds the best model(s) or reaches the computation limitations.

4 EXTENDED MODEL SPACE SPECIFICATION

4.1 Agent Property Specification

Our previous work assumes relatively simple scenarios where only several reserved properties are considered for each agent. To support more complex scenarios, we would allow a modeler to add new properties into the model space for automated model discovery. The goal of property specification is to provide a well-defined structure for agent properties so that new user-defined properties can be added into the model space and be searched by the search method in a unified way. To achieve this goal, a formal and general structure for user-defined properties is needed. We define a user-defined property **p** has a general structure as below:

p = <type, range, $\mathbf{v_{init}}$> where

Type: numerical, or categorical

Range: if type is *numerical*, range =[$\mathbf{v_{low}}$, $\mathbf{v_{upper}}$], where $\mathbf{v_{low}}$ is the lower bound of the numerical value, and $\mathbf{v_{upper}}$ is the upper bound of the numerical value.

if type is *categorical*, range = the set of all possible categorical values.

$\mathbf{v_{init}}$**: initial value** of the property:

if type is *numerical*: $\mathbf{v_{init}}$ is a real number between $\mathbf{v_{low}}$ and $\mathbf{v_{upper}}$

if type is *categorical*: $\mathbf{v_{init}}$ is an element of the **Range** set.

To add a new property to the model space, a modeler needs to specify the type, range, and $\mathbf{v_{init}}$ for that property so that it can be searched by the search method. Typically, the modeler has some knowledge about the system and the property, and thus can define the type and range of a new property to be searched. For example, if agents' energy is important for a specific application, the modeler can define a new property called energy, and specify its type to be numerical and define its range to be between 0 and 100. Similarly, if agents can change color between green, yellow, and red for a specific application. The modeler can define a property called color that has **type** = categorical and **range** = {green, yellow, red}. The modeler may also specify the initial value $\mathbf{v_{init}}$ for the property if starting from that initial value is important for the application. Otherwise, by default $\mathbf{v_{init}}$ is a random value within the Range.

We note that specifying the type, property, and $\mathbf{v_{init}}$ of a property is different from defining how the property is used by specific behaviors. The modeler specifies the property, but then it is up to the search method during the model discovery process to decide if and how the property will be used by a specific behavior. In general, when a modeler provides a property that is meaningful and has relevant range, it would make it easier for the search method to find behaviors using this property.

4.2 Activation Component

In previous work, the described actions from the multiple behaviors of a same behavior group is averaged to compute the final action. In many cases, this method does not work well because the desire action of each behavior would cancel each other after being averaged. For example, Figure 3.a shows a scenario where agents have an obstacle avoidance behavior but cannot successfully perform it. Figure 3.b explains why this happens. In this example, agent A has two behaviors that manipulate its direction: B1 and B2. Agent A uses behavior **B1** to avoid the obstacle by steering to the left, and **B2** to follow its nearest agent. Because the final decision $\overline{\boldsymbol{B}}$ is averaged between **B1** and **B2**, agent A moves toward to the obstacle directly.

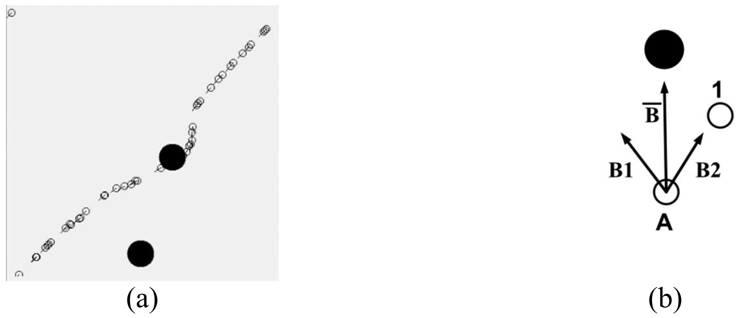
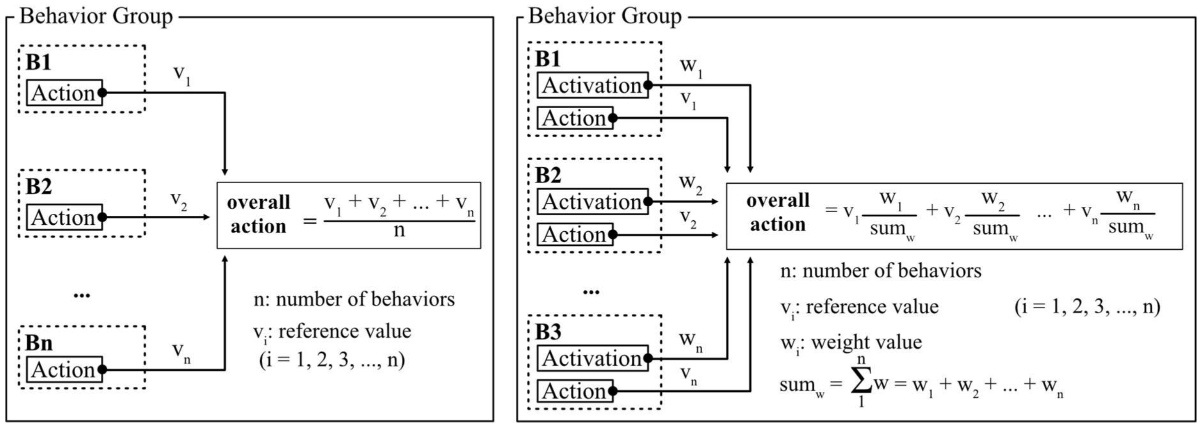


Figure 3: An example where average method does not work well.

To address this problem, we need a mechanism to specify the importance of each behavior at each iteration. Therefore, we extend the previous work so that each behavior has two components: an Activation component and an Action component. The Activation component specifies the level of activation of the behavior. This allows the priority of a behavior to be modeled because different activation levels represent different priorities. The Action component specifies the action of the behavior, i.e., how the behavior changes the value of a property. This component is the same as a behavior in previous work. In other words, previous work considered only the Action component, and our extension adds a new Activation component.

With the differentiation of the Activation component and the Action component, Figure 4 shows how the overall action of a behavior group is computed in previous work (4.a) and in current work (4.b). As can be seen, in previous work, each behavior B_i returns a reference value $\mathbf{v_i}$ (i = 1, 2, 3, …, n), then the final result is averaged among all reference values.

In the current work, along with reference value v_i, each behavior B_i also returns a weight value w_i. The overall action is the summation of product between normalized weight $\frac{w_i}{sum_w}$ and reference value v_i of each behavior. Because this weight distribution method sums up all $\frac{w_i}{sum_w}$ to 1 after normalization, the more weight a behavior has, the more portion it takes. Hence, this is an approach to support the priority mechanism.



(a): Average overall action. (b): Weight overall action.

Figure 4: Combining desired actions from multiple behaviors.

4.2.1 Activation Component Specification

The steps of the Action component are similar to the steps of the behavior in the previous work (illustrated in Figure 1). Thus, in this paper we focus on the Activation component. In general, the Activation component checks the condition of itself or its environment and returns a weight represents the importance of the behavior. This involves sensing the environment based on FOV, applying filters, and extracting property (referred to as activation property) in the same way as in the Action component. Afterwards, it uses an activation function to compute a weight based on the activation property value. Figure 5 illustrates the steps of the Activation component, which returns an activation weight in the end. Note that a special case is to check an agent's own property to computer the activation weight. In this case, a self-filter is used to return the agent itself before the extra property step.

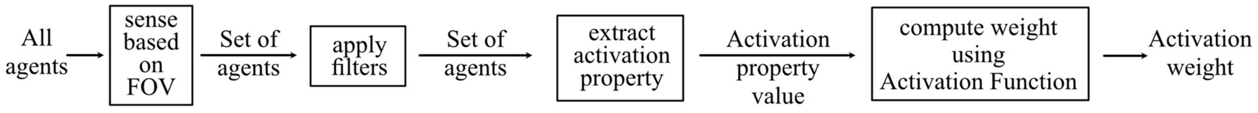

All agents
sense based on FOV
Set of agents
apply filters
Set of agents
extract activation property
Activation property value
compute weight using Activation Function
Activation weight

Figure 5: Steps of the activation component.

More formally, we define the Activation component to have three elements:

Activation = <$\mathbf{F_a}$, $\mathbf{p_a}$, activation_function>

$\mathbf{F_a}$ is a set of filters where $\mathbf{f_a} \in \mathbf{F_a}$

$\mathbf{f_a}$ is a filter where:

$\mathbf{f_a}$ = <$\mathbf{p_f}$, $\mathbf{c_f}$>

$\mathbf{p_f}$ is the filtered property

$\mathbf{c_f}$ is the filter criteria:

- if $\mathbf{p_f}$ is *numerical* type, $\mathbf{c_f} = [c_{f\_low}, c_{f\_upper}]$, where v_{low} is the lower bound, and v_{upper} is the upper bound of the criteria.
- if $\mathbf{p_f}$ *categorical* type, criteria = a subset of the set of all possible categorical values.

$\mathbf{p_a}$ is the activation property.

activation_function = <**type**, $\mathbf{r_a}$> where:

type: activation type: binary or linear (see explanation below).

$\mathbf{r_a}$ is the activation function range, where $\mathbf{r_a} = [r_{a\_low}, r_{a\_upper}]$, follows the same regulation as $\mathbf{c_f}$.

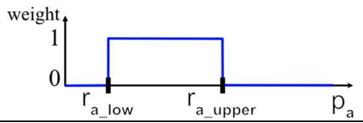
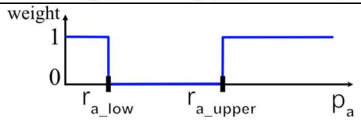
The activation function computes a weight based on the value of the activation property. This function needs to have a well-defined structure so that it can use the same structure to cover different situations of changing activation weight based on dynamic values of the activation property. In this work, we allow a behavior to have one of the following two activation function **types**: **binary** function and **linear** function. **Binary** function returns weight value of 1 or 0 based on if p_a is inside or outside a specified range $[r_{a\_low}, r_{a\_upper}]$. By giving weight of 0, a behavior is considered not activated because its product is also equal 0 and not contributed to the overall action. In some cases, modelers want the passing conditions are not within the range. To capture all these situations, specification of the binary function is:

binary = <inside> where**:**

inside: a Boolean value to decide the satisfied conditions are inside or outside the range of $\mathbf{r_a}$

Table 1 shows the details of the binary function. When **inside** is True, the function returns weight of 1 if $\mathbf{p_a}$ is within the range of $\mathbf{r_a}$, and 0 otherwise. When **inside** is False, the function returns 0 if $\mathbf{p_a}$ is within the range of $\mathbf{r_a}$, and 1 otherwise.

Table 1: Weight value using binary function.

Inside	**Weight value**	**Function graph**
True	$weight = \begin{cases} 1\ if\ p_a \in [r_{a\_low}, r_{a\_upper}] \\ 0\ if\ p_a \notin [r_{a\_low}, r_{a\_upper}] \end{cases}$	
False	$weight = \begin{cases} 0\ if\ p_a \in [r_{a\_low}, r_{a\_upper}] \\ 1\ if\ p_a \notin [r_{a\_low}, r_{a\_upper}] \end{cases}$	

Different from the binary function that returns only two values: 0 or 1, the linear function returns different values based on the inputs of the activation property. The specification of linear function is:

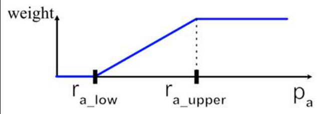
linear = <slope, increase> where:

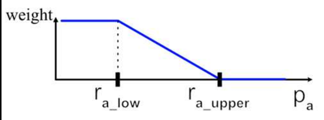
slope: measures the rate of weight change for different property values**.** It is an integer and has range $[0, s_{upper}]$. s_{upper} is the upper bound of **slope** value.

increase: a Boolean value to indicate the increment/decrement direction of the weight value.

To make the search process more efficiency, modelers need to define upper bound of **slope**. Thus, the search space includes all possible values within range of $[0, s_{upper}]$ (note: in our implementation we define a granularity to make the search space finite). Table 2 illustrates the linear function for various **slope** and **increase** variables. If **increase** is True, the weight increases when $\mathbf{p_a}$ moves toward to the upper bound $\mathbf{r_{a\_upper}}$ of $\mathbf{r_a}$. If **increase** is False, the weight increases when $\mathbf{p_a}$ moves toward to the lower bound $\mathbf{r_{a\_low}}$ of $\mathbf{r_a}$. When $\mathbf{p_a}$ is no longer within range of $\mathbf{r_a}$, the weigh is equal to either **0** or ($\mathbf{r_{a\_upper}} \times$ slope) depends on the $\mathbf{p_a}$ and **increase** value.

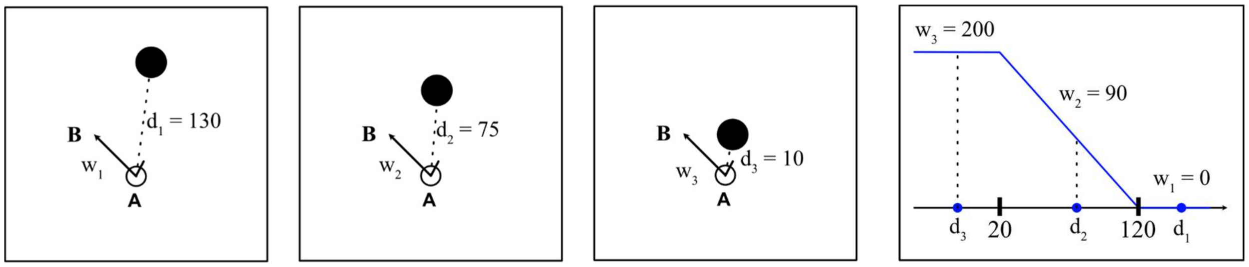
Table 2: Weight value using linear function.

Increase	**Weight value**	**Function graph**
True	$weigh = \begin{cases} 0 & if\ \boldsymbol{p_a} < r_{a\_low} \\ \boldsymbol{slope} \times (\boldsymbol{p_a} - r_{a\_low}) & if\ r_{a\_low} \le \boldsymbol{p_a} \le r_{a\_upper} \\ \boldsymbol{slope} \times (r_{a\_upper} - r_{a\_low}) & if\ \boldsymbol{p_a} > r_{a\_upper} \end{cases}$	

False	$weight = \begin{cases} \boldsymbol{slope} \times (r_{a\_upper} - r_{a\_low}) & if\ \boldsymbol{p_a} < r_{a\_low} \\ \boldsymbol{slope} \times (r_{a\_upper} - \boldsymbol{p_a}) & if\ r_{a\_low} \leq \boldsymbol{p_a} \leq r_{a\_upper} \\ 0 & if\ \boldsymbol{p_a} > r_{a\_upper} \end{cases}$	

4.2.2 Illustrative Example

Below is an illustrative example shows how the activation weight of a behavior is computed at different time steps. Figure 6 shows this example, where an obstacle is moving toward to agent A and the distances between them at time t = 0, 10 and 20 are 130, 75, and 10 respectively.



(a) t = 5 (b) t = 10 (c) t = 15 (d) weight assignments

Figure 6: A scenario where obstacle is moving toward to an agent at three different time steps and their weight values.

Behvaior B changes the agent's direction if it is too close to an obstacle. The specification of behavior B is:

Activation:
Filter:
f_1 : **p_f** = type
c_f = obstacle
p_a = position
r_a = [20-120]
(note: distance range between 20 and 120).
Activation function:
type: linear, **increase** = False, **slope** = 2.

Action:
Filter:
f_1 : **p_f** = type
c_f = obstacle
p_{action} = position (converted to the direction towards that position and use it as the reference value).
offset: 40 (add 40 degree to the reference direction)

Behavior B's activation has one filter with filter property = *type* and criteria = *obstacle*. In other word, the filter chooses only obstacles and eliminates regular agents. Because **p_a** chooses position as activation property, the linear activation function calculates the activation weight based on range **c_a** = [20,120] and the relative distance between position of agent A and position of the obstacle. Figure 6.d shows how the weight is calculated. Because **increase** value is False, weight is decreasing when distance is increasing. In this example, Figure 6.b shows the distance d2 = 75 and it within activation criteria range. As a result, weight of behavior B at t = 10 is 2 × (120-75) = 90. At t = 5 and t = 15, because the distance d_1 (Figure 6.a) and d_3 (Figure 6.c) both exceed the range of activation criteria **c_a**, weight of B at t = 5 equals to 0 (d_1 = 130 > **c_{a_upper}**), and weight of B at t = 15 is 2 × (120 – 20) = 200 (d_3 = 10 < **c_{a_low}**). For the Action component of this behavior, it has one filter that is similar to the filter of activation. It then extracts the obstacle's position to get the direction to the obstacle as the reference value. A 40 degree is added to this reference direction to make the agent turn away from the obstacle.

5 EXPERIMENTS

5.1 Snake formation with Patience

To demonstrate adding user-defined properties for automated model discovery, we consider an example that we studied in our previous work: snake shape (Table 4 - ID: 4) with personal space (Table 4 – ID: 5,6) (Keller and Hu 2019). In that example, agents form a snake shape and also maintain personal space, so they do not collide with each other. The discovered model works well if number of agents and world size are set correctly. However, in the situation where there are too many agents in a limited space world, there is not enough space for agents to execute the speeding up behavior. As a result, their speeds decrease to zero and stand still until the rest of the simulation (Figure 7.a). To prevent that from happening, beside snake formation and maintain personal space, we also add speed goal. Function s$peedzero(a_i, t)$ (1) adds penalty every time speed of agent a_i is zero at a specific time step t by returning value of 1. Equation (2) sums all the penalty during the whole simulation. Hence, the lower the sum value is, the better the speed goal achieves.

$$speedzero(a_i, t) = if\ (speed(a_i, t) = 0)\ return\ 1, else\ return\ 0. \tag{1}$$

$$Number\ of\ speed\ is\ zero\ = \sum_{t=0}^{T} \sum_{i=0}^{|A|} speedzero(a_i, t) \tag{2}$$

T is total number of simulation iterations.
|A| is a set of agents, (a_i,t) is the ith agent at time step t.

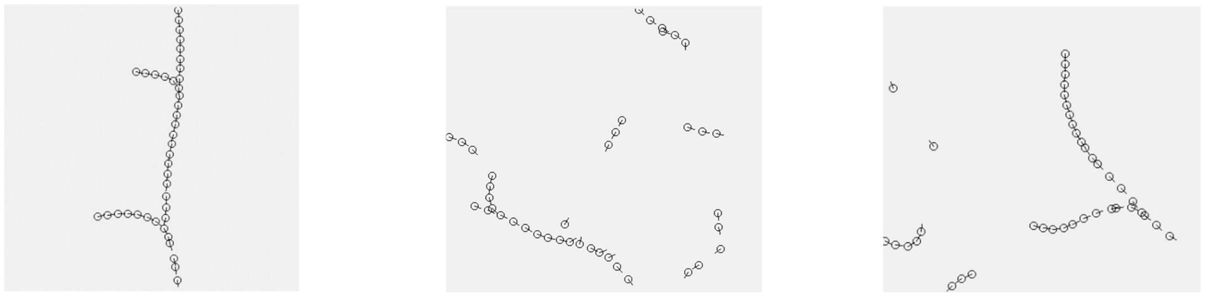
With this speed fitness function added, a better model is discovered and agents able to maintain acceptable speeds. The behavior to keep the speed is when speed of an agent is in range [0,0.1], it turns left 60 degree. Hence, open more spaces for agents to speed up. However, agents cannot maintain a stable snake formation because most of the agents change their directions too quickly as Figure 7.b shows.

To find models that can lead to stable snake formation, it is necessary to add a new property that can control how long agents need to wait before changing the direction. Thus, we add a user-defined property called **patience** to the search space. Patience is numerical type with initial value $\mathbf{v_{init}}$ = 100 and range = [0,100]. Table 3 shows the final discovered model specification, and the first three (ID: 1-3) are used by agents to wait patiently before changing their directions.

Table 3. A set of behaviors to maintain more stable snake formation.

Behavior Group: Patience		
ID	**Behavior specification**	**Purpose**
1	**Activation:** if self-speed is within [0, 0.1], return 1. **Action:** reduce the value of self-patience by 1.	If speed of agent itself is slow, its patience decreases.
2	**Activation:** if self-speed is within [0.2, 2], return 1. **Action:** increase the value of self-patience by 2.	If speed of agent itself is fast, its patience increases.
Behavior Group: Angle		
3	**Activation:** if self-patience is within [5,10], return 1. **Action:** direction increases by 50	If patience of agent itself is low, it steers left 50 degree.
4	**Activation:** if having one or more regular agent within FOV, return 1. **Action:** change direction to nearest agent	Follow the nearest agent.
Behavior Group: Speed		
5	**Activation:** if the distance to nearest agent is within [0,15], return 1 **Action:** speed decrease by 0.3	Slow down if too close to an agent.
6	**Activation:** if the distance to nearest agent is within [20,150], return 1 **Action:** speed increases by 0.5	Speed up when there is enough space ahead.

Whenever speed of an agent decreases and within range [0 – 0.1], it begins to lose patience by 1 at each iteration. During that time, if agent is able to gain speed above 0.2, its patience increases by 2. If not, the agent will wait until its patience level is below 10, then turn its direction 50 degree to the left. In other word, instead of breaking the snake shape immediately, agents wait until their patience is low. As a result, these behaviors maintain snake formation (Figure 7.c) better than previous one. Without a user-defined property, it is challenging to express a behavior that makes agent wait patiently before making another decision.



(a) (b) (c)

Figure 7: Three experiments of snake formation with personal space.

To compare quantitative results between previous and current experiment, three fitness functions are used include: snake shape, personal space, and speed fitness function. First, snake shape fitness function measures the differences between an agent's direction and the direction toward to its nearest neighbor. Second, personal space fitness function adds penalty whenever agent violate other agents' personal space (Keller and Hu 2019). Next, speed fitness function is calculated using equation (1) and (2). Final fitness score of them are averaged between 100 simulations. Lastly, the scores are normalized to have common scale of 0 being the best, and 1 being the worst. Compare the results between experiment 1 (without **patience** property) and 2 (with **patience** property) of Table 4, we see that experiment 2 gives better score in snake shape and personal space behaviors because the snake shape maintains stably for a longer time. Hence, agents do not change direction as frequency as experiment 1 and have less chance to collide with each other. Speed in other hand, experiment 1 receives the better score because agents in experiment 2 will wait an amount of time before changing the direction when their speed is zero. Thus, more speed penalties are added to it.

Table 4. Fitness scores of experiments with and without patience.

Snake Shape + Personal Space + Maintain speed models	
Experiment 1: without Patience property	**Experiment 2:** with Patience property
Snake Shape: 0.77 Personal Space: 0.513 Speed Fitness: 0.12	Snake Shape: 0.53 Personal Space: 0.308 Speed Fitness: 0.22

5.2 Snake formation with obstacle avoidance

A snake formation and obstacle avoidance model was discovered in previous work. However, it works well only if obstacles are static. In other word, obstacles' position does not change during the simulation. When obstacles can move around, agents often collide with them directly (Figure 3.a). With the extended model space specification that includes the **Activation** component in behavior, a new model was discovered that allows agents to form snake shape while avoid moving obstacles. Table 5 presents the new obstacle avoidance behavior. This new obstacle avoidance behavior has a linear activation function. If distance between an agent and its nearest obstacle is larger than 120, the weight is 0 and the behavior make no impact to the final decision. The closer the distance, the higher the weight is. The maximum activation weight 12

× (120 – 80) = 480. Compare to snake shape behavior (similar to behavior of Table 3 – ID: 4) that returns weight of 0 or 1, obstacle avoidance behavior weight can heavily affect the final outcome.

Table 5. Obstacle avoidance behavior with linear activation function

Behavior Group: Angle		
ID	**Behavior specification**	**Purpose**
1	**Activation:** if there is an obstacle within FOV, return a weight follows below conditions: $$weight = \begin{cases} \mathbf{12} \times (\mathbf{120} - \mathbf{80}) & if\ \boldsymbol{dist} < 80 \\ \mathbf{12} \times (\mathbf{120} - \boldsymbol{dist}) & if\ 80 \leq \boldsymbol{dist} \leq 120 \\ \mathbf{0} & if\ \boldsymbol{dist} > 120 \end{cases}$$ **dist:** relative distance from agent to its nearest obstacle. **Action:** direction increases by 20	Agent steers left 20 degree when there is a nearby obstacle. The weight has range [0,480]. The closer the distance between the agent and the obstacle, the higher the weight is.

Figure 8.a illustrates agents are forming a snake line and there are two obstacle moves around. At this moment, the obstacle is far away, so the priority to avoid it is still low. Later on, for a group of agents, the priority to avoid the obstacle increases when there is one moves toward to them (Figure 8.b) Eventually, obstacle avoidance behavior becomes dominant, and agents ignore the snake formation behavior. Hence, agents are able to avoid the coming obstacle completely (Figure 8.c).

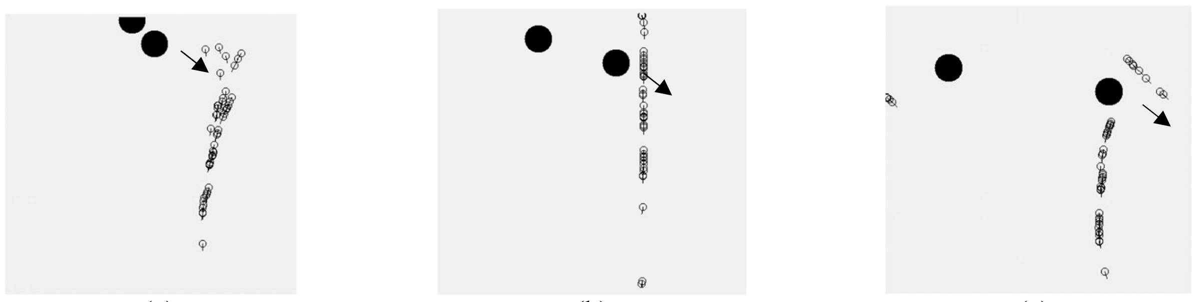

(a) (b) (c)

Figure 8: Snake shape and obstacle avoidance with priority at different time step.

Snake shape and obstacle avoidance fitness function are used to compare quantitative results between experiment 1 (without priority) and 2 (with priority). Similar to maintain personal space fitness function, obstacle avoidance function increase penalty by 1 every time agents touch obstacles at each iteration. (Keller and Hu 2019). The results of Table 5 show that model of experiment 2 is better. It is predictable because for experiment 2, depend on surrounding environment, either snake shape or obstacle avoidance behavior are more important. Hence, final overall movements mostly influence by one of them, and emergence phenomena are shown clearer.

Table 5. Fitness scores of experiments with and without priority.

Snake shape + Obstacle avoidance models	
Experiment 1: Without priority	**Experiment 2:** With priority
Snake Shape: 0.22 Obstacle avoidance: 0.687	Snake Shape: 0.153 Obstacle avoidance 0.041

6. CONCLUSION

This work extends the model space specification for automated model discovery by adding **user-define property** and **Activation** component to make the framework support more complex applications. Experiments with and without patience property clearly show the necessary of user-define property. With a formal specification provided, modelers can to add new suitable properties to the search space. Hence, archive the better solution for their modeling tasks. Activation component decides which agents' behaviors has highest priority depends on nearby neighbor and environment. Thus, the framework assists better for complex models where agents need to compute overall movement action from multiple behaviors. Future works include: first, improve the efficiency of the searching process by reducing search time and discovering better models. Second, compare the end results with other approaches such as: handcraft model or models that are generated by different search algorithms. Third, apply the framework on more complex model tasks; such as circle formation with obstacle avoidance to validate the robustness of the system.

REFERENCES

Beck, E. C., M. Birkett, B. Armbruster, B. Mustanski. 2015. "A Data-Driven Simulation of HIV Spread Among Young Men Who Have Sex With Men: Role of Age and Race Mixing and STIs". *JAIDS Journal of Acquired Immune Deficiency Syndromes* 70(2):186-194.

Bhouri, N., B. Flavien, and P. Suzanne. 2012. "An Agent-based Computational Approach for Urban Traffic Regulation". In *Progress in Artificial Intelligence,* edited by M. Díaz and S. Ventura, 139-147. New york, New York: Springer Nature.

Decraene, J., M. Low, F. Zend, S. Zhou, and W. Cai. 2010. "Automated Modeling and Analysis of Agent-based Simulations Using the CASE Framework". In *11th International Conference on Control Automation Robotics & Vision*, December 7th-10th, Singapore, Singapore, 346-351.

Deeter, K., K. Singh, S. Wilson, L. Filipozzi, S. Vuong. 2004. "APHIDS: A Mobile Agent-Based Programmable Hybrid Intrusion Detection System". In *Mobility Aware Technologies and Applications,* edited by A. Karmounch, L.Korba, E. Madeira, 244-253. Berlin, Heidelberg: Springer.

Antonini, G., M. Bierlaire, and M. Weber. 2004. "Discrete Choice Models of Pedestrian Behavior". ROSO Reposrt 040916, Operations Reseach Group ROSO, Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland.

Horne, G. E., and T. Meyer. 2004. "Data Farming: Discovering Surprise". In *Proceedings of the 2004 Winter Simulation Conference*, edited by R. R. Ingalls, M. D. Rossetti, J. S. Smith, B. A. Peters, 813-818. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Huang, Y., and A. Verbraeck. 2004. "A dynamic data-driven approach for rail transport system simulation". In *Proceedings of the 2009 Winter Simulation Conference*, edited by M. Rossetti, R. R. Hill, B. Johansson, 2553-2562. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Keller, N., and H. Xiaolin, 2019. "Towards Data-Driven Simulation Modeling for Mobile Agent-Based Systems". *ACM Transactions on Modeling and Computer Simulation* 29(1):1-26.

Kendall, E. A., P. V. M. Krishna, and C. B. Suresh. 2000. "An Application Framework for Intelligent and Mobile Agents". *ACM Computing Surveys* 32(1es):20-es

Lee, K. H., M. G. Choi, Q. Hong, and J. Lee. 2007. "Group Behavior from Video: A Data-Driven Approach to Crowd Simulation". In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, August 3rd-4th, Diego, California, 109-118.

Reynolds, C. W., 1987. "Flocks, Herds and Schools: A Distributed Behavioral Model". *ACM SIGGRAPH Computer Graphics* 21(4):25-34

Wagner, N., and V. Agrawal. 2014. "An Agent-based Simulation System for Concert Venue Crowd Evacuation Modeling in the Presence of A Fire Disaster". *Expert Systems with Applications: An International Journal* 41(6):2807-2815.

Whitley, D. 1994. "A Genetic Algorithm Tutorial". *Statistics and Computing* 4:65-85.

AUTHOR BIOGRAPHIES

HAI LE is a PhD student in the Computer science at Georgia State University at Atlanta. He received his master's degree from the University of Central Arkansas. His current research focuses on Agent-Based simulation and modeling, particularly on model specifications. His email address is: hle49@student.gsu.edu.

XIAOLIN HU is a Professor of the Computer Science Department at Georgia State University, Atlanta, GA. He received his Ph.D. from the University of Arizona, Tucson, in 2004. His research interests include modeling and simulation theory and application, complex systems science, agent and multi-agent systems, and advanced computing in parallel and cloud environments. His work covers both fundamental research and applications of computer modeling and simulation. He was a National Science Foundation (NSF) CAREER Award recipient. His email address is: xhu@gsu.edu.