A Visual Approach for Modeling Agent Intelligence

Philip Jensen, Ye Liu, Alejandro Lopez-Lago, Aaron Welch
Advisor: Dr. Mark Lewis
30 July 2009

Abstract

With the increasing trend towards multi-agent simulations in social science research, it has become important to be able to easily and effectively design intelligent agents to meet the requirements of the experiments. However, the experimenters are generally not proficient in programming, so there is demand for simple interfaces for visually designing agents. We have created an interface that diagrams the flow of data in an agent's decision making process. The implementation of this involves routines containing a set of input and output tabs that guarantee the type-safety of connections to other routines. This system has been integrated with the existing CASE environment, and we have built simple simulations using this program. With this system social scientists can intuitively create agent intelligences for their simulations.

Introduction

The CASE (Cognitive Agents for Social Environments) system is very versatile in that it allows many different types of simulations to be run within its framework. The user has the ability to define the number of agents, the actions of the agents, and many other aspects of simulations. In addition, once defined, these simulations are easily reusable and do not require re-programming of many properties. However, a major drawback of this setup is the creation of new simulations. Setting up a valid simulation requires the user to be fluent in the Java programming language, and be fairly familiar with the underlying framework of the CASE system. This is not a trivial requirement, and many social scientists, and others who may want to use the system for their own simulations, simply do not have the skill set needed.

We strive to create a graphical interface for CASE to overcome this drawback. Since our interface for the CASE environment essentially needs to graphically represent some programming logic, we begin by examining common graphical representations of programming logic.

One common graphical representation of programming logic is in the form of data flow diagrams. These diagrams use a series of shapes to represent the operation to be performed, and different types of lines to show the direction of data flow. Data flow diagrams are useful due to how much can be expressed with a few simple objects. However, its complexity increases drastically for larger programs, and the diagram can quickly become obscure and difficult to understand. To overcome some of the complexity of extreme size and interconnectedness, the basic diagram may be grouped into larger conceptual pieces, so that each piece can be modeled individually. Another visual system used for designing programs is entity-relationship models. Entity-relationship models (E-R models) layout how entities are related to their actions and data members. An example of this could be a “has” relationship, whereas an entity is linked by a “has” block to each data element present within the entity. While these models are easy to read, they are not well-suited for writing programs since they become messy as they scale up and can

contain ambiguities in the relationships between agents and actions.

Therefore, it is necessary to evaluate ways in which these methods may be combined to create an interface that is as easy to use as it is powerful. Instead of using the aforementioned diagrams to represent agent logic, a more abstract design would lead to diagrams being more understandable and concise. Common functionality and learning processes can be represented as single objects to make the diagram conceptually simple. However, there is also a tradeoff between conciseness and generality; a design that is overly abstract may become too limiting for general usage or for those who have some programming background. Some possible approaches to the graphical design of an agent creation interface are analyzed in the following sections, from which we extract some ideas to incorporate into our own model.

Related Work

Current multi-agent simulation tools that contain a visual creation interface include programs such as Multi-Agent Modeling Language (MAML) and Zeus. MAML, created by László Gulyás, Tamás Kozsik, and Sándor Fazekas, is a tool for social scientists lacking programming knowledge to create multi-agent simulations. MAML contains a list of icons that represent different objects in the environment to make it easier to create a simulation. For example, an agent is shown as a single person, a group of agents is shown as a group of people, and an equation is shown as an icon with the text "x=." However, MAML still forces the user to use a scripting language to design the simulations; although one does not need to be as proficient in programming as if they were using a non-graphical interface, one still needs to be able to fully understand scripting languages.

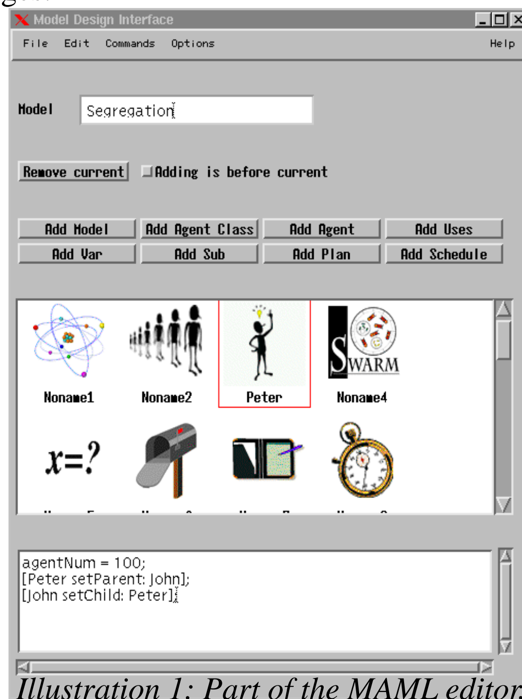

Illustration 1: Part of the MAML editor.

Another popular tool is Zeus, a cross-platform and distributed multi-agent simulation tool created by Hyacinth Nwana, Divine Ndumu, and Lyndon Lee. Simulations can be created completely inside the visual editor. However, the lack of instructions makes it difficult to install on Linux environments, and the program does not display properly on Windows Vista. There has not been updates on the program and it is fairly outdated.

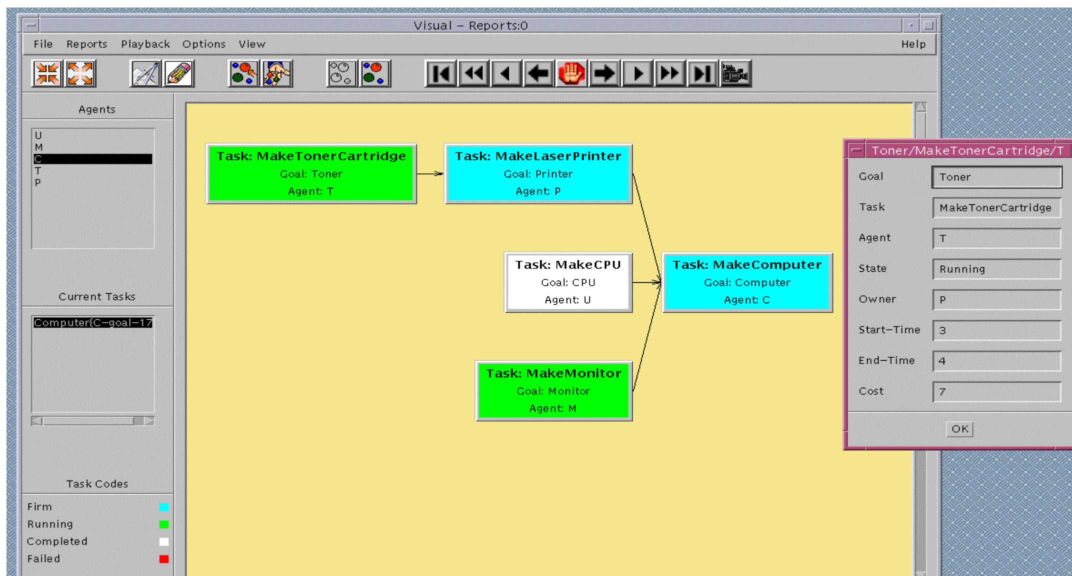

Illustration 2: An example simulation in the Zeus environment.

Visualization programs have also implemented successful visual interfaces for manipulating data to create graphics. SwiftVIS, a creation of Mark C. Lewis, uses a series of data-filters to convert data into graphs. SwiftVIS uses different colors for different types of objects, and it also contains objects that act like “sinks”: they do not produce outputs. Another visualization tool with a novel and effective interface is IBM's OpenDX. OpenDX also bases its editor environment on a data flow model, but its objects contain input and output tabs representing the necessary connections for each object. This allows for data to be better organized.

Problem Definition

Through this project, we hope to create a front-end user interface for CASE that is graphic-based, easy to understand, and easy to operate for users who do not have background in computer programming. There are several problems that we must overcome in order to achieve this goal. First, our interface must be general enough so that a variety of simulations and agents can be created, but must also not become overly complex to use. There is an inherent trade-off between flexibility and complexity; the more flexible and general a program is, the more the user must work to put basic functions together and create complex functions. We must decide on how simple our basic functions will be: for example, is “move” too restrictive? Is “divide” too simple? Another problem that we have to tackle is how to actually run the diagram. The methods for accomplishing this could be one of two things: translate the diagram into code, or just parse the diagram at runtime. Since the interface aims to be flexible, we must take this into account when we put in the basic outline of the underlying code. If we were to create Java code for each diagram, simple compile-time errors would be easily found, but we could risk creating overly general code that may not be adequate for all applications. However, if we just parse the diagram at run time, the system would be more flexible in that it would be able to run anything the user creates, but would lack many features to prevent program crashes.

Although there have been similar graphical interface programs, our project is unique in that it is not simply a graphical diagram or flowchart. It will have significant underlying code; each agent diagram represents the decision-making process of an agent, the overall diagram represents the environmental setup, and the program must combine the two diagrams and CASE. In other

words, our project will have three levels: agent, environment, and underlying code, and it will be fully graphically programmable on the first two levels.

Our Agent Intelligence Builder

There are many features of our AI builder that make it stand out from existing graphical programs used for the same purposes. One of the most unique aspects is the use of routines and subroutines within the system. Each routine is an encapsulated unit that accepts inputs and returns outputs obtained by carrying out certain operations on the inputs. There are two kinds of routines: primitives, which are non-user-editable; and user-defined, which can be customized to fit the specific requirements of a simulation. This separation of editable and non-editable routines gives our program the power to be very flexible, yet at the same time fit for abstraction, due to the fact that primitives reduce the need for repeated and tedious building of common, standard operation routines, while editable routines free users from the restriction of having only pre-defined operations. Editable routines can contain other routines, which helps de-clutter the user interface by allowing users to edit their program on higher levels and abstracting routines. Primitives are currently provided by us only, while user-defined routines can be created by the user in any way that they see fit. Currently implemented primitives are detailed in the next section.

Another feature of our system is the use of type-checking and cycle detection for directing data flow. Each routine has tabs for inputs and outputs, which are typed. When a tab is selected, all other tabs whose type match and thus can be connected to the selected tab are highlighted. Connecting to non-compatible tabs are not allowed, reducing the probability that users would misdirect their data to routines that do not process them correctly. There is also a topological sort/cycle detector implemented in the system. All routines are added to a graph that is topologically sorted to be processed in the right order, so that those that rely on the output of others will not be evaluated before their due time. As an extra feature, the topological sort also detects cycles in the routine graph and prevents users from unwittingly creating infinite loops in their agent logic.

Our program also accommodates the needs of users who have some computer programming background by including “reflect formula” support. All formula inputs can be simple numbers, expressions, or more complicated reflect formula, which are somewhat like code snippets. Currently, many inputs require the use of reflect formulas, but we hope to decrease the dependence on reflect formulas so that the program would be more user-friendly.

When the user has determined that an agent's physical state needs to be changed, such as its location in space, changes need to be made that affect the system at a higher level than the agent alone. This is so that the server knows the new information about the agent's location and status. As such, agents can perform *actions*, which are changes of state that directly affect the environment. There are currently four actions that agents can make: the Move action moves the current agent a specified delta-x and delta-y distance away, the MoveTo action sets the location of the agent in the environment to the specified x and y coordinates, the Die action removes the current agent from the simulation, and the Spawn action creates a given agent at a location determined by a x and y coordinate.

In addition to being passed into routines from previous time steps, inputs can also be passed in by reflect formulas and manual input. This can be done through a menu item in the overall window, and is especially helpful in testing certain qualities of a program, such as the use of randomly-generated numbers.

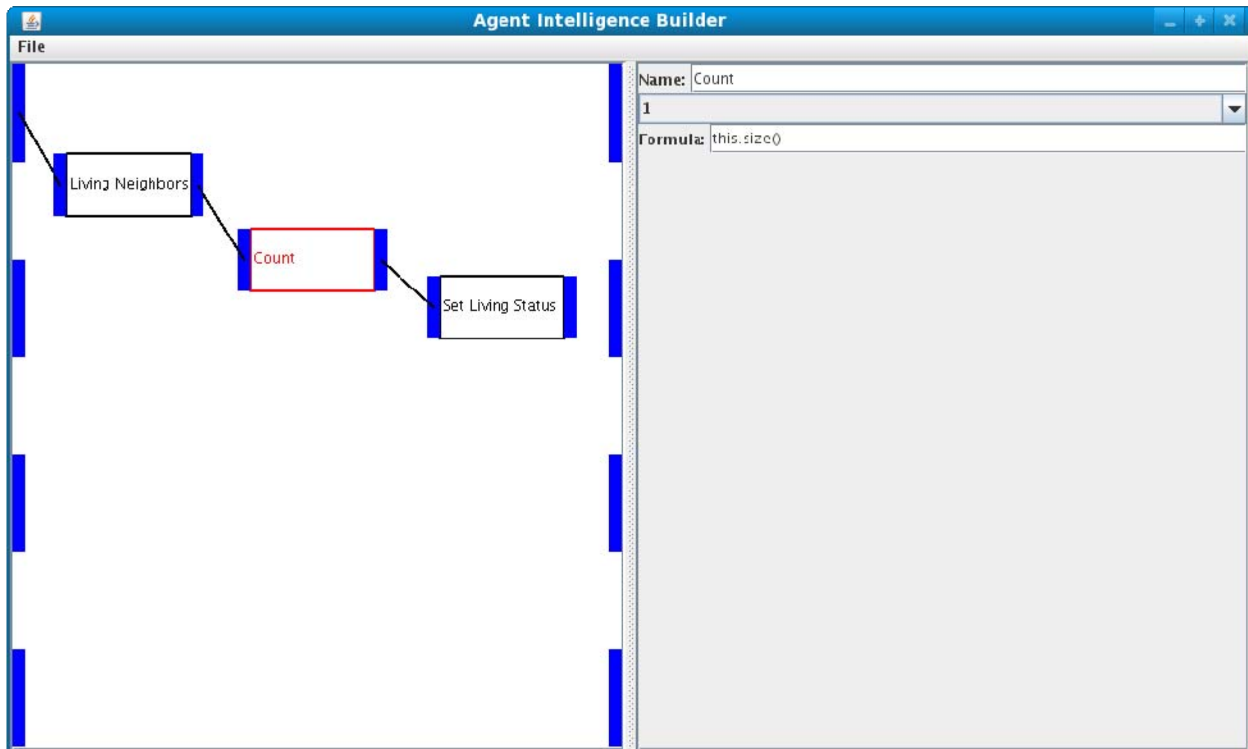

Illustration 3: The interface for testing agent intelligences. Each list represents *one of the input tabs to the main routine. When run, the data passed to* the output tabs and the entity's variables are displayed.

Implemented Primitives

The Reduce routine acts like the reduce or fold function found in many functional programming languages. Given an initial value, a function to apply to the input list, and the name given to the currently reduced value in the given function, the reduce routine will apply the function to the currently reduced value and the next element in the list until the end of the list. The final reduced value is sent out as an output. The reduce routine is useful for computing properties about a list, such as the sum or product of a sequence of numbers.

The Select routine acts like the filter function in functional languages. The routine takes in a list of data and adds the elements that agree with the select formula to the outgoing list. For example, if the select formula was “this > 2”, then the outgoing list would be the list of elements from the input list that are greater than 2. One of the uses for this routine is for selecting a specific type of agent to work on from the neighbors tab.

The Map routine changes all of the elements in the input list by using a specified formula and returns a new list with the modified elements in the same order; it is the equivalent of the map function in functional languages. It is useful for modifying agents in a list; for example, incrementing a counter that an agent may have.

The Min/Max routine serves as a filter that selects the maximum or minimum value from the given input, which should be a list with members that can be compared between each other. The routine takes a single input, which contains a list of members, and has a single output, which contains either a list containing the selected member, or a null list if the input list was empty. The

option of selecting the minimum or maximum is determined by checking a radio button on the options panel.

The If routine allows for selection of different outputs or actions based upon multiple possible states which the user is able to specify. These states can be declared sequentially in the form of an if-elseif-else structure, each evaluating a user-specified boolean statement based upon the input data. Each condition is associated with a particular response that determines what kind of output will result from the execution of the routine. The first condition to evaluate as true will carry out the desired response from the user. The routine has the ability to add, remove, and reorder any number of these conditions, such that they can ensure proper handling of data.

The Sort routine takes in any number of comparable objects as inputs and sorts them sequentially. Currently it can only sort in "ascending" order, but in the future, it can be made to support more complicated sorting.

The List Map routine allows the user to operate upon lists themselves rather than individual elements in order to gain properties of the list such as its size. This is useful for things such as counting the number of neighbours for an agent, as is needed in the Game of Life.

Example/ Analysis

To put our system to the test, we developed an implementation of Conway's Game of Life using routines. Agents, or cells are evenly spaced on both the x and y axes, creating a grid-like layout. The Game of Life starts off with an initial configuration consisting of non-moving cells that vary only by whether or not they are considered alive or dead. For each step thereafter, whether or not a particular cell is alive or dead depends entirely upon the number of alive cells directly surrounding it. If a particular cell is dead, then it becomes alive if there are exactly three live cells around it. However, if a cell is alive, then it dies if it has less than two or more than three live cells around it, and remains alive if there are exactly two or three live cells. The point of this is to create complex emergent patterns from a simple set of rules.

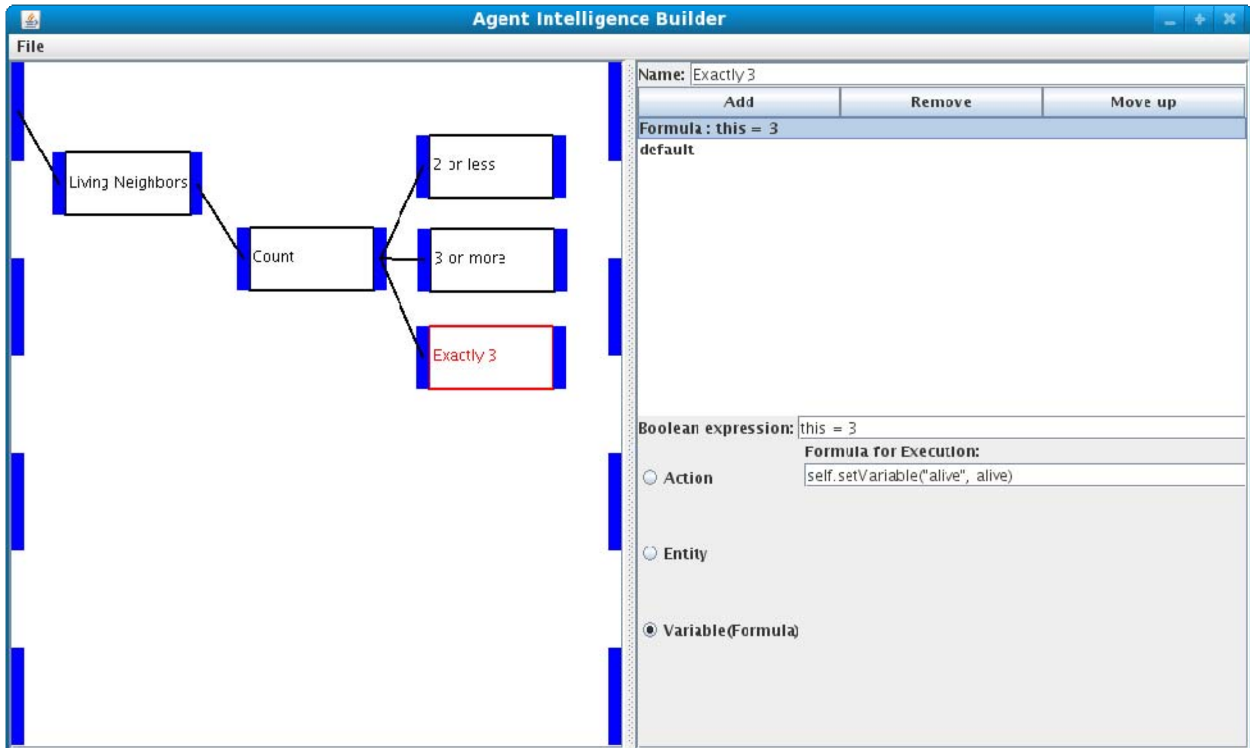

Illustration 4: Agent intelligence for the Game of Life. The first routine gets all of the living neighbors and the second routine counts them. The third routine then decides if the current agent is dead or alive.

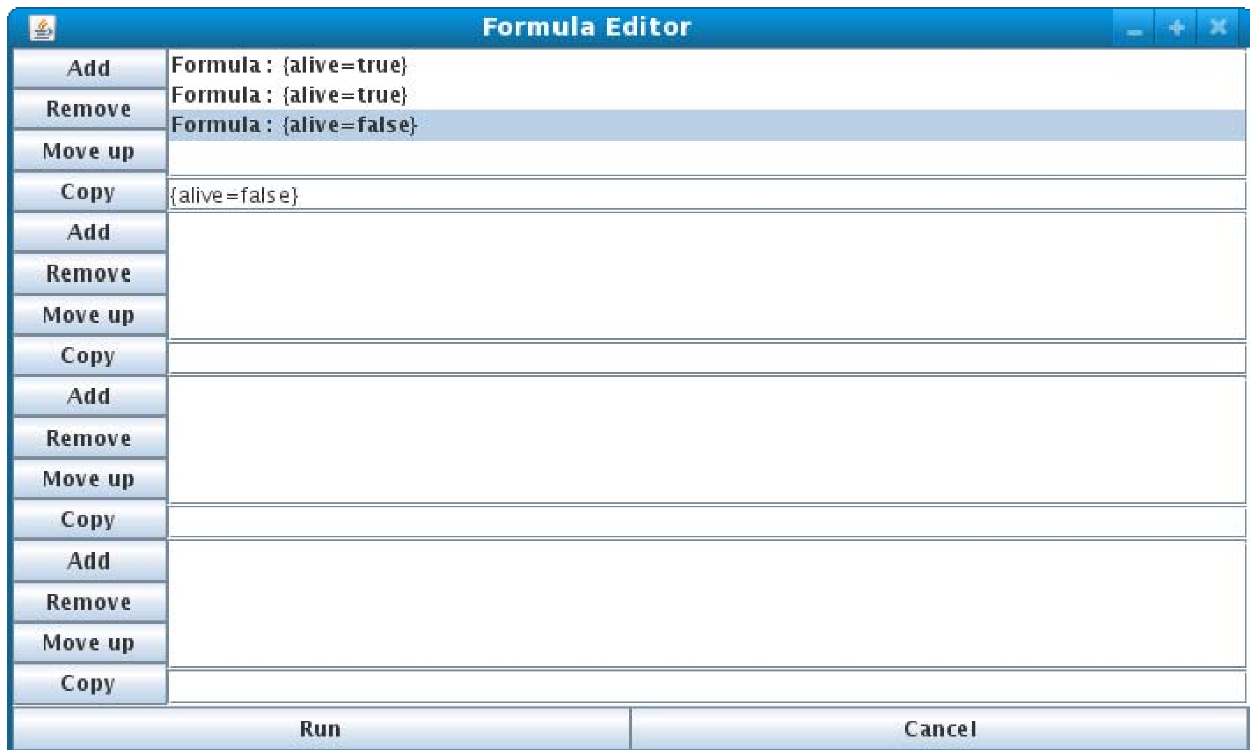

Illustration 5: Another way of writing agent intelligence for Game of Life. In this case the conditions are split up into three distinct cases. The Count routine sends its information to all three if-routines.

Creating this simulation using routines involved adding a filter (select routine) that checks the alive state of neighboring entities and returns the live cells. The processed cells are then sent to a list map routine that counts the number of cells in the list. This number is then passed to a series of if statements that checks for whether or not the cell is alive and the live neighbors is less than two or greater than three or if the cell is dead and the live neighbors are exactly three. If one of these statements evaluates to true, the alive state of the agent gets changed by a variable such that if it was alive before, it is dead after, or vice versa.

This simulation provides a good demonstration of how simply the agent intelligence can be created using this new interface. We expect that implementing sugarscape would be similarly simple, and as such that would likely be the next design attempt using this system. There is still a little code-style logic involved in the creation of reflect formulae, but this could be simplified for the average user in future versions of the code. Overall, the design of this graphical interface appears to be heading in a positive direction.

Conclusion

The current build is powerful enough to create the Game of Life using only three generic primitive routines. This indicates that building basic simulations with the program is feasible and simple. However, the current program still requires a large amount of Java method calls to create a non-trivial simulation; this can be fixed by creating either more primitives or prepackaged user routines for common tasks such as counting the elements in a list or retrieving a variable. The program can also include routines for common machine-learning techniques such as Q-learning or neural networks. The type-checking can be made more strict by creating primitive routines that input and output subtypes of Object; this would prevent errors caused by using a method that the object doesn't support in a routine (i.e. this.size() if this was an Integer). The routine-based entity also needs to improve compatibility with visualization software such as SwiftVis. Finally, the interface can be improved by integrating it with the simulation creator and by allowing users to import previously created routines. The simulation editor should also include more methods for placing agents, such as an interactive grid where users can create agents on the locations they select. Thus, while major improvements are still needed, this program has the potential to be a powerful and intuitive tool for creating complex simulations.