Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 00 (2013) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2013

Nova: A modern platform for system dynamics, spatial, and agent-based modeling

Richard M. Salter[a,*]

[a] *Computer Science Department, Oberlin College, Oberlin, OH 44074, USA*
Current Address: ESPM Department, University of California at Berkeley, Berkeley, CA,

Abstract

In this paper we describe *Nova*, a new Java-based modeling platform that naturally supports the creation of models in the system dynamics, spatial and agent-based modeling paradigms. *Nova* uses a visual language to express model design, and provides automatic conversion for such models to script form for execution. *Nova*'s architecture promotes hierarchical design, code reuse, and extensibility through the use of plug-ins. The Nova Website, `www.novamodeler.com`, is being built to foster a vibrant user community by providing ample support for model and plug-in construction, and user services such as online repositories for user-contributed content.

Keywords: modeling, simulation, system dynamics, agent-based models, cellular automata

1. Introduction

The rate at which emerging computational fields, particularly those in the biological sciences, can progress depends upon the existence of powerful, but easy to use, computational platforms. These platforms should be able to facilitate the training of students with little programming experience, as well as carrying out heavy duty simulations as part of research agendas. According to [1], modeling "tends to be splintered across numerous disciplines. ... Moreover, models have a variety of representations." Many of today's most popular simulation tools, such as Stella [2], VenSim [3], Berkeley Madonna [4] and NetLogo [5] support only a particular narrow design paradigm: Stella, Madonna and VenSim system are dynamics-based and use the well-known "stock and flow" flowchart language [6]. NetLogo is an extension of the Logo programming language [7] to implement spatial and agent-based models.[1] Other tools, such as MatLab [8] and the statistical language *R*, [9], while powerful and well-designed, tend to be applied in an ad-hoc fashion to individual projects; consequently they do not promote much code reuse.

In this paper we describe *Nova* [10], a new Java-based modeling platform that naturally supports the creation of models in the system dynamics, spatial and agent-based modeling paradigms in a single desktop application by introducing a unifying design concept. *Nova* supports both visual and scripting languages and is intended to be both an educational and research grade tool.

*Corresponding author. Tel.: +0-440-775-8095 ; fax: +0-440-775-6638. Supported in part by NSF grant CNS-0939153.
E-mail address: rms@cs.oberlin.edu.

[1] NetLogo does provide a second platform for stock-and-flow type simulations, but it is completely separate from the agent-based platform and only shares input/output components with the latter.

Nova is fundamentally a dynamic modeling system that is extended through hierarchical design to express spatial and agent-based architectures. A *Nova* model can be built using the visual language, and then by using its *capture* function be automatically converted into a runnable script[2] for immediate execution, or possible deployment over a network or on a supercomputer.

Nova focuses on the creation of a modular unit called a *capsule*. Each capsule is a complete model that interacts with its environment through an interface consisting of input and output channels. The simplest capsule might contain a stock-and-flow model similar to one built in Stella. However, capsule instances (called *chips*) may appear in other capsules (as long as there is no circularity), communicating with their hosts through their I/O channels. Each chip introduces into its host the functionality of that chip's encapsulated model. Capsules may also be exported and reused in other projects.

Spatial and agent-based models are constructed using array-like data structures called *aggregators*. The current implementation provides three aggregator types::

agent vectors are one dimensional arrays of *agents*; an agent is a capsule "wrapper" that includes a representation for location and movement within a two dimensional space. Agent vectors also manage dynamic creation and destruction of agents.

cell matrices are two dimensional arrays of capsules. They provide a means for representing cellular automata.

sim worlds combine agent vectors with cell matrices, so that agent locations correspond to matrix coordinates. The result is a virtual space of interacting agents and cells.

Additional spatial aggregators, for three dimensional and non-Euclidean spaces such as networks, are under development.

Nova's computational architecture comprise the semantics of *NovaScript* [3], a scripting language embedded in Javascript. The NovaScript runtime environment is an extension of the ECMA 1.7 Javascript standard. All *Nova* simulations are actually NovaScript programs executing on the NovaScript runtime environment.

In the next section we discuss *Nova*'s architectural design. Section 3 provides a brief look at NovaScript, and Section 4 shows examples using the visual language. Conclusions and future work are presented in Section 5.

2. *Nova* Architecture

We begin the description of *Nova* with a look at the platform architecture. This architecture is represented in both the visual and NovaScript languages. The capture tool converts the visual representation into a script.

2.1. Simulators

A simulation in *Nova* is a sequence of state transitions from a start time to an end time using a fixed time increment dt.[4] A *Nova simulator* is an object containing all of the required structure to produce a simulation except for the timing element. When combined with a *clock* and set to an initial state, a simulator executes to produce a final state. We use the term *strobe* for each iteration step produced by the clock.

The current set of simulator types includes the *capsule* (the simplest complete simulator) and the three aggregators described above. Figure 1 illustrates the structure of simulator types. The top-level simulator is always a capsule. A strobe received by a capsule containing an aggregator is passed to the aggregator, which distributes the strobe to its components. Distributed strobes may process in parallel, to be synchronized at the next strobe.

[2] In this regard *Nova* resembles the relationship between Simulink [11] and MatLab

[3] NovaScript has similarities with the Modelica model specification language [12], however unlike Modelica NovaScript is intended to be an execution model

[4] Variable step-size methods will also be made available at some future date.

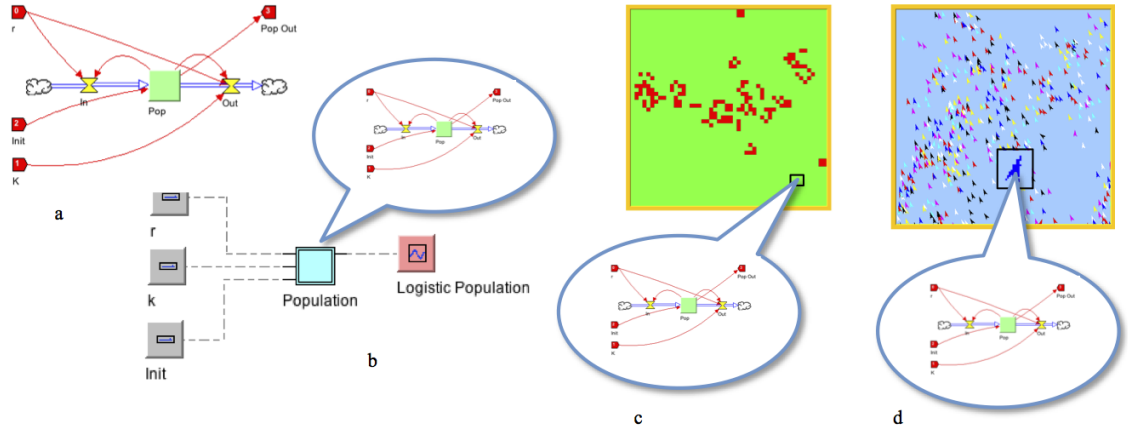

Fig. 1. Simulator structure: a) capsule representing a complete system dynamics model; b) embedding of a capsule in a chip for use at a higher level; c) embedding of a capsule as an element of a cellular automaton; d) embedding of a capsule as an agent in simulated world.

2.2. Primitive Components

Capsules are comprised of interacting components which fall into two categories: *stateful* or *stateless*. A stateful component keeps track of its value over time while a stateless component is only aware of its current value. At each strobe, the new values of stateful components are computed using their current and past values.

Nova operates in either continuous or discrete mode. In continuous mode state change specifications are treated as the derivatives of continuous functions, hence new values are computed using numerical integration. *Nova* currently supports three integration methods: RK (Runge-Kutta) 4, RK 2 or Euler. In discrete mode, *dt* is set to 1 and the Euler method is used.

The most familiar stateful component to users of Stella and similar systems is the *stock*. Stock state changes are based on the values that flow in and out through *flow* components connected to the stock. *Nova* adds two additional primitive stateful components:

variable A variable may be used when the state change can be expressed succinctly through a derivative function. It is the equivalent of having a stock with a single bi-directional inflow.

sequence Sequences may be used in place of stocks in discrete mode. In this case, rather than expressing the next state in terms of a difference, the next state is specified directly using a "next" property.

In addition to flows, *Nova* provides the *term* component as a place for specifying functions of the current state.[5] For convenience there is also a *command* component, used to carry out any necessary meta-state change to the execution environment.

2.3. Clocks

As one would expect, clocks are responsible for synchronized iteration of the simulation process. Execution is carried out by associating a clock with a simulator. Clocks are defined using four parameters: start and end times, *dt* and the selected integration method. The clock initializes the simulation and strobes its associated simulator at each iteration.

Nova defines a system clock to control the run of the top-level simulator. Additional clocks associated with component simulators turn out to be useful; see Section 2.5 below.

[5] Stella calls such components *converters*.

2.4. Chips

A *chip* encapsulates an instance of a capsule to act as a *submodel* within a host capsule. More than one chip containing instances of the same submodel, or different submodels, may appear in a single capsule. Large *Nova* models can be built using stacks of submodels based on a well-structured design pattern. Submodel capsules that have general applicability can be saved and reused in multiple projects. The use of submodels, in chips and aggregators, is a principal feature of *Nova*'s design architecture.

2.5. Clocked Chips

One variation on the basic chip structure is the *clocked chip*. A clocked chip is a chip that has been provided with its own clock. Each strobe of the host clock causes the clocked chip to do a complete run using the parameters of its own clock. *Nova* uses this to implement synchronized clocks with multiple levels of time granularity. Clocked chips are particularly useful as a way to stage multiple runs with different parameters, for sensitivity analysis and other purposes.

2.6. Code Chips

Another chip type, the *code chip*, provides a means of extending the visual language with scripted methods. Each code chip contains a NovaScript method (i.e. in the form of a JavaScript function). The code chip represents an invocation of that method within the flow diagram. The code chip integrates the method's inputs and outputs through connections to the rest of the flow diagram. Code chips may compute a set of data outputs or define a function usable in other components. Code chips are also exportable for reuse in other projects. A library of code chips of commonly used functions for a given domain can be created and shared among applications.

2.7. Aggregators

The aggregators create virtual spatial environments in which individual capsules function as cells or agents. Each capsule component of an aggregator is provided with methods that access aspects of the aggregator environment. For example, each cell in a cell matrix can access the entire matrix; consequently a cell may determine a set of neighbors and query state values in those neighbors. Similarly, the agent vector provides methods for each agent to interact with every other agent. A sim world adds methods facilitating communication between cells and agents.

2.8. Controls and Displays

A *Nova* simulation is expected to interact with input controls, such as sliders, and output displays, such as graphs and tables. A third type of device, the *plug-in*, is a generic I/O component that provides enhanced visual displays and other extensions to the basic architecture. We refer to control and display components as *dashboard components*. In NovaScript, we use proxies to as intermediaries between NovaScript simulation objects and dashboard components. During execution, the proxies are bound to actual dashboard control and visualizing elements.

Among the plug-ins already implemented are ones to visualize cell- and agent-based domains, as well as to compute histograms and tabulate averages. *Nova* has a well-defined plug-in API so that users with knowledge of Java can create domain-specific plug-ins and add them to their *Nova* environment.

2.9. Component Properties

Component properties define the model of interest. Stocks, variables and sequences all have an initial value property; a variable also has a *prime* property defining its derivative. A sequence has a *next* property defining its next value. Flows and terms have *expression* properties defining their current values. Properties are algebraic terms over floating point numbers and component names, where a component name is used to represent the current value of that component. See Sections 3.1 and 3.2 for examples.

2.10. Methods

Methods are Javascript functions linked to a particular capsule. A method may contain expressions referring to any component in the capsule for its current value. A particularly expressive format for defining methods within the visual language is through the use of code chips. See Section 2.6 and Section 4.2 for an example.

3. NovaScript

In this section we show some examples of how this architecture is used by NovaScript to express a system dynamics model. NovaScript is embedded in and extends the ECMA 1.7 Javascript standard. As a consequence of this embedding, NovaScript code is syntactically structured using Javascript data and program structures. A complete description of Javascript can be found in [13]. NovaScript uses Javascript's limited vocabulary of data types and structures to express the *Nova* architecture.

The Javascript datatypes are as follows:

Data types	floating point numbers strings	`1, -20, 3.1415927` `"this is a string", 'so is this'`
Data structures	arrays	`[1,2,3]`
	generic objects	`{first_name: "John", last_name: "Doe", age: 45}`
Program structure	functions	`function(x,y,z){return x + y - z;}`

A Javascript variable `x` is declared using

```
var x = ...
```

and can be assigned to any of these values.

The only novel data structure in this list is the generic object. In Javascript, this simplest object is an association list of property names with property values, with no additional semantics. Object components are accessed either using dot or array notation; e.g., `x.first_name` or `x["first_name"]`.

NovaScript extends Javascript by exposing within a Javascript environment special object types implemented in Java. These object types include capsules, stocks, terms, cell matrices, agent vectors, sim worlds, etc. The resulting objects have more complex internal structures than the generic object defined simply by a property list, however they can be used within a NovaScript program like any native Javascript object.

As an example consider a simple exponential population model in which the population grows at a rate proportional to its current value. A stock-and-flow model for this example would include a stock to hold the current population, a flow to define the change in population, and perhaps a term to hold the constant of proportionality. There are several ways in which this model can be defined in NovaScript.

3.1. Direct Definition of a Simulation

The *Nova* model will consist of a capsule containing a variable and a term. (We can substitute a variable for stock and flow given the simplicity of the model.) Capsule instances implementing particular models can be created directly in NovaScript, as shown below, by defining a constructor function for that model. Our constructor, called `PopCapsuleBuilder`, takes 3 arguments: a name for the model, a starting population, and a growth rate. This constructor calls NovaScript constructors for objects `Variable` and `Term`[6] to respectively create new instances for components bound to the variables `pop` and `rate`.[7] The model is programmed by setting component properties. Variable `pop` has its `initial` property set to the `start` parameter passed to `PopCapsuleBuilder`. `pop.prime` is assigned a string expression that is the derivative of population growth in this model. (The *verbose* property causes each step of the simulation to be printed.) This expression contains a reference to the `rate` term whose `exp` property is assigned to the argument `r`; consequently it will evaluate to `r` whenever `rate` is evaluated. Having assigned these properties, the capsule is created by passing the list of components to the `Capsule` constructor, which returns an executable simulator for the model. This becomes the result of `PopCapsuleBuilder`.

```
var PopCapsuleBuilder = function (name, start, r) {
    var pop = new Variable("pop");
    var rate = new Term("rate");
    pop.initial = start;
    pop.prime = "rate * pop";
```

[6] In Javascript the `new` operator is used with a constructor function to create object instances.

[7] NovaScript objects all require a name string as the first parameter to the constructor.

```
    rate.exp = r;
    pop.verbose = true;
    var ans = new Capsule(name, [pop, rate]);
    return ans;
};
```

To execute the model, we create an instance of the capsule, passing in an initial value and growth rate, respectively, of 1 and 0.1. We also require a clock instance with appropriate parameters: our model will run from 0 to 100 with a *dt* of 0.1 using the RK4 integrator. Because we set the `verbose` property to `true` in the constructor, the population for each step of the simulation is printed.

```
js> var pop = new PopCapsuleBuilder("pop", 1, 0.1);
js> var cl = new Clock(0, 100, 0.1, "RK4", true);
js> cl.init(pop)
js> cl.run()
  0.100 pop        1.01005
  0.200 pop        1.0202010866708333
  ....
 99.900 pop        21805.490675434194
100.000 pop        22024.63767384653
```

Note that our constructor can be used to create multiple capsules with different initial and growth rate parameters to be run independently.

3.2. Projects and Schemas

Large NovaScript programs, including those created by the capture mechanism, are built as *projects*. A project bundles together a set of interacting definitions to produce a complete runnable model. Key to this ability is the use of *schemas*, which are a central design feature of NovaScript. A schema is a Javascript object that serves as the class definition for creating the objects used by the simulation. Schemas are used to specify each type of simulator (capsule, agent vector, cell matrix, sim world); they also specify the control, display and plug-in proxies that serve as intermediaries between a running NovaScript program and the the user interface.

A schema definition is a generic Javascript object listing a set of properties. Schemas use two properties in their definition to specify the relationships in the model being built. First, the schema property `settings` is a generic object whose name/property-value format lists input/output connections between pairs of components. A second schema property, `dynamics`, is a list of equations in the form of an array of strings. Each equation defines some property of a simulation component. This format has the advantage of reflecting the equations that define model conceptually. For example, in the population model, an equation such as `pop.prime = rate * pop` is drawn from the underlying differential equation defining this model.

Each schema also contains a `specifies` property that describe the type of the schema. There are also properties for each component type (e.g., `stocks`, `flows`, `terms`, `variables`, etc.); and any external displays, controls or plugins.

Here is a capsule schema for the same population model described in Section 3.1; however in this version we have added several dashboard components: a slider called `birth_rate` for determining the value of `r`, and table and graph components `poptable` and `popgraph` for viewing the results of the simulation. Note that the dashboard components appear in a property list in which each named object is associated with a type. This type is actually another schema. For example, in the assertion `poptable: "main_poptable"` the *object* called `poptable` is an instance of the object specified by the `main_poptable` schema. The latter defines a proxy for a visible table component on the *Nova* dashboard. Proxy schemas such as `main_poptable` use the `proxy` property to bind the proxy to an actual dashboard component. (i.e., `birth_rate`, `poptable`, and `popgraph`.)

```
SimplePopModel.defineSchema('main', {
    specifies: "CAPSULE",
    variables: 'pop',
    terms: 'rate',
    controls : {birth_rate: "main_birth_rate"},
```

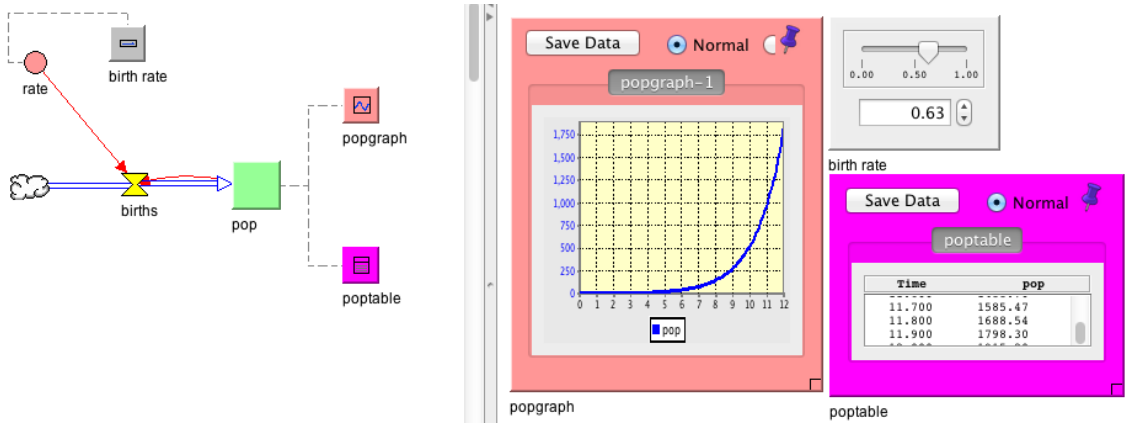

Fig. 2. *Nova* visual representation of a simple population model

```
    displays: {poptable: "main_poptable", popgraph: "main_popgraph",},
    dynamics: ['pop.initial = 1', 'pop.prime = pop * rate', 'rate = birth_rate'],
})

// Displays & Controls

SimplePopModel.defineSchema('main_birth_rate', {specifies:"Slider", proxy:"birth_rate",});
SimplePopModel.defineSchema('main_poptable', {specifies:"Table", proxy:"poptable"});
SimplePopModel.defineSchema('main_popgraph', {specifies:"Graph", proxy:"popgraph"});
```

4. Using the Visual Language

The *Nova* visual language greatly facilitates construction of a *Nova* simulation. Such a simulation is modeled as a set of flow diagrams built on a canvas out of appropriately connected visual components. Each flow diagram is equivalent to a single capsule.

Figure 2 shows the visual source of the simple population model in Section 3.2. *Nova* will use the *capture* process to translate this visual depiction of the model into a project containing the code in Section 3.2. In Figure 2, the left pane, the *model canvas*, contains the model's flow diagram. The right pane, the *dashboard*, contains controls and data visualizing components. (These are the `proxy` components bound in the Section 3.2 schemas). Each dashboard component is represented by a small tile on the model canvas, showing its role in the flow diagram.

In the remainder of this section we show some more advanced examples illustrating *Nova*'s expressiveness.

4.1. Multilevel System Dynamics Models using Clocked Capsules

Figure 3a shows a capsule containing a logistic population model with fixed parameters, constructed similarly to the example in Figure 2. This model runs from 0 to 100 time units with a *dt* of 0.1. In Figure 3b, we have created a second capsule containing a clocked chip. The chip's encapsulated model is the logistic population model of Figure 3a (as indicated by the balloon) and its clock is parametrized identically to the top-level clock used in that simulation. We have added components that increment the carrying capacity K from 50 to 70 in unit steps. Running the top-level clock in Figure 3b produces the result shown in the graph: each step in the value of K produces a complete run of the logistic population simulation.

The generality of the clocked capsule means that the user can construct any desired pattern of parametrization of a sequence of multiple runs. One usually finds such batch or multiple run facilities built into the execution environment of simulation software rather than reflected into the programming environment; consequently they are limited in their ability to express different patterns of execution.

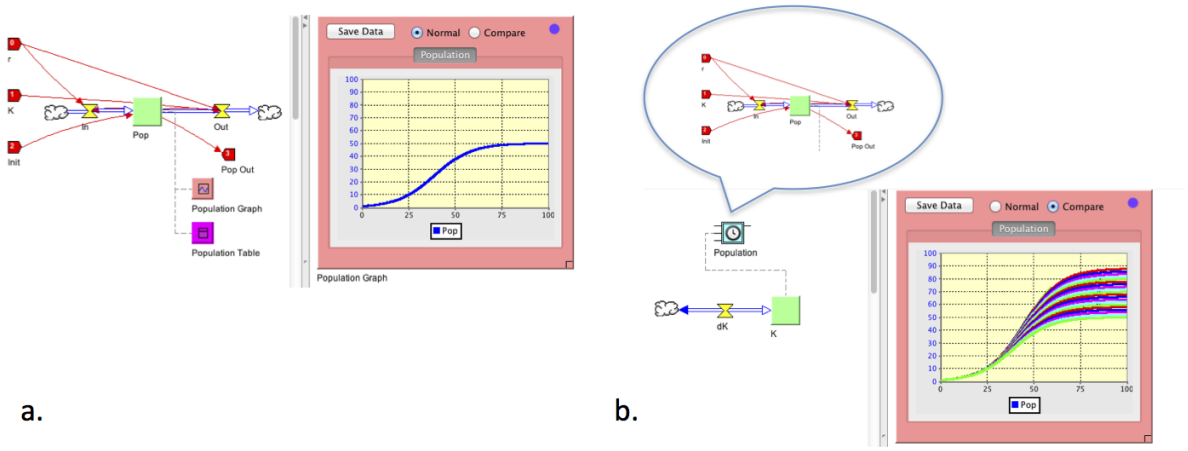

Fig. 3. (a) logistic population model; (b) using clocked capsule for sensitivity analysis.

4.2. *Conway's Game of Life*

A popular illustration of cellular automata is *Conway's Game of Life* [14]. Implementation in *Nova* requires an $n \times n$ cell matrix containing n^2 instances of a capsule built to maintain the state of a single cell, and to transition that state according to the rules of the game. Figure 4.2a shows the flow diagram defining the capsule of a single cell. According to the rules of the Life game, each cell's next state is determined by its current state and by the state of the 8 neighbors comprising its Moore neighborhood.[8] This simulation will only be possible if each cell can query its neighbors to determine the transition.

Nova provides an efficient way to accomplish this. Before the simulation starts each cell builds an array of references to its neighbors. The code for this is shown below:

```
ans = [];
for (var i = -1; i <= 1; i++)
    for (var j = -1; j <= 1; j++) {
        if (i == 0 && j == 0) continue;
        var x0 = mycoords.row+i; var y0 = mycoords.col+j;
        if (x0 >= 0 && x0 < rows && y0 >= 0 && y0 < cols) ans.push(Super.matrix[x0][y0]);
    }
```

Here the free variables `rows` and `cols` indicate the dimensions of the matrix, while `mycoords` is bound to the coordinates of the cell executing the call. `Super.matrix` is a reference to entire the matrix of cells. The code selects those cells that are within 1 unit of the current cell in either the horizontal or vertical direction and stores them in an array. The code chip `neighbors_1` contains this code – it inputs the coordinates of the calling cell from the term `mycoords` and passes the computed array to the term `nbrs`. When it comes time to compute the next state of this cell, each element, x, of the array of neighbors can have its state read as the value of x.`alive`. The code for computing the next state is contained in the code chip `nextstate_1`, which inputs the current state (`alive`) and the array of neighbors (`nbrs`) to compute its result. Note that just by looking at its structure, this flow diagram shows that state transitions rely on a combination of the the cell's current state and that of its neighbors.

In Figure 4.2b we see the top-level capsule containing the 50×50 cell matrix `Life_Matrix`. The latter is connected to the plug-in `board` that displays the current state of the matrix. We add a third code chip `cellcount_1` containing code to count the number of living cells and show that value in the plug-in `alive`.

4.3. *Agent-Based Examples*

We briefly describe three advanced agent-based models implemented in *Nova*. Please visit the *Nova* Website, `www.novamodeler.com` for further details.

[8] In the next generation, a dead cell surrounded by exactly 3 neighbors lives, and a live state not surrounded by either 2 or 3 neighbors dies.

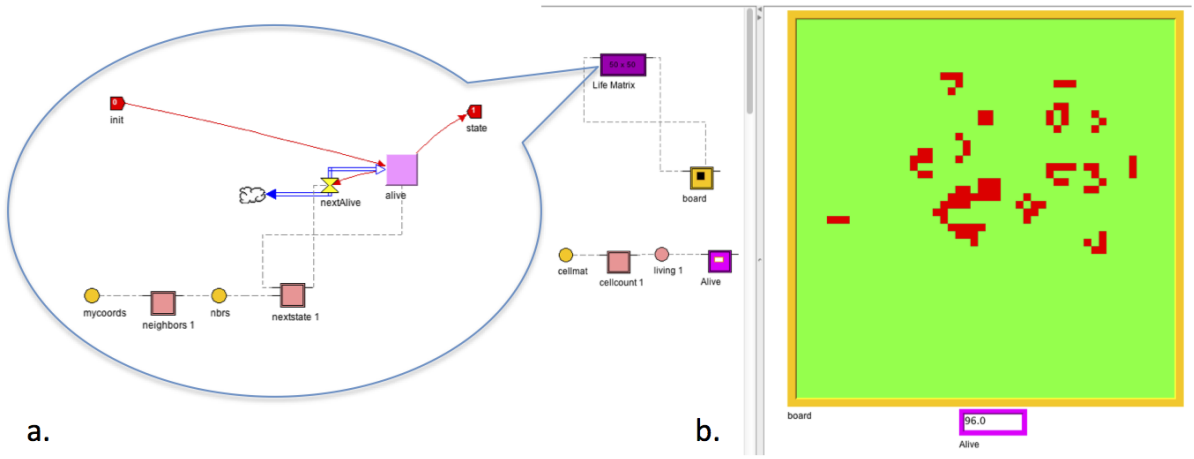

Fig. 4. (a) *Life* cell capsule; (b) *Life* using a cell matrix and plug-in display.

4.3.1. Moths

This example shows an agent-based (though totally artificial) model of moth-like behavior. In this environment cells have a dynamically changing temperature, and moths are attracted to the warmest cells. This model's implementation shows how cells and agents interact. Moths behave by flying at constant speed in a direction that may change at each iteration. The chosen direction is one of three, either straight ahead or at a 45^{o} angle left or right. The chosen direction is the warmest of the three. At the beginning, all of the heat is concentrated in a square in the center of the matrix. Heat diffuses from a cell to each neighbor at a rate proportional to the temperature difference between the pair.

The *Nova* implementation uses a sim world at the top level to hold arrays of moth and cell capsules. Each moth capsule uses a standard submodel to implement its forward motion. A moth can find out the temperature of any of the cells it is near, and so at each iteration the moth will test the temperature of the appropriate cells in each of the three available directions. Meanwhile, diffusion continues to change the temperature in the cells. Figure 5a shows the display at about halfway through the simulation. Eventually moth behavior becomes random when the diffusion evens out the temperature throughout the matrix.

4.3.2. Flock

Flock (Figure 5b) is based on the NetLogo implementation [15] of the *Boids* model [16] of emergent behavior. The individual agents initially move in random directions but are ultimately organized to move in a single direction. This example requires direct agent-to-agent interaction.

4.3.3. Ants

Figure 5c is an artificial example of ant behavior, also based on a NetLogo model [17]. Ants are attracted from their nest in the center to food sources, and then to return with food to the nest. When holding food they drop a pheromone that attracts other ants to their trails. Ants may die anywhere but are born only in the nest.

5. Conclusion / Future Work

The Nova Website, `www.novamodeler.com`, is being built to foster a vibrant user community by providing ample support for model and plug-in construction, and user services such as online repositories for user-contributed models, plug-ins, code chips, etc. Beyond its self-contained power and utility, the ability of Nova to attract such a user community will be greatly enhanced through facilities to communicate, as seamlessly and rapidly as possible, with other computational platforms. Of particular importance are the back and front end computational process that organize data to be used by Nova and analyze data generated as a consequence of Nova simulations. Nova is already able to interface with the statistical and data analysis packages *R* [9], to post

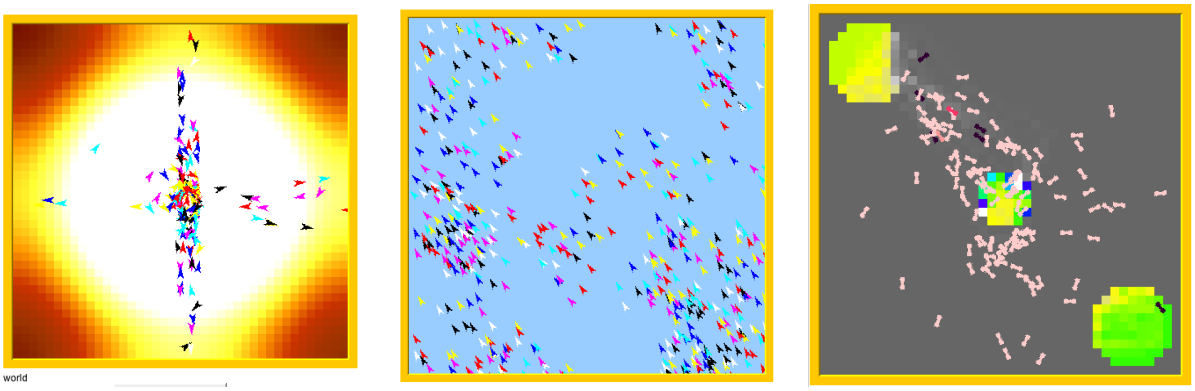

Fig. 5. (a) *Moths*; (b) *Flock*; (c) *Ants*

process and analyze data generated during Nova simulations. In the context of computational population biology, for example, Nova will be extended to collect input from geographic information systems (GIS) software, such as proprietary ArcGIS [18] and open source GRASS [19] platforms. These platforms are used to manage and manipulate raster and vector-based mapping and landscape information that Nova can leverage for simulations of agents moving over real, informationally detailed landscapes. Nova will then compute interactions between and among agents and landscape factors, resulting in changes to both agent and landscape states. This opens the door to a range of modeling projects based on real-world landscape data derived from remote sensing imagery or spatial analyses, as well as facilitate iterative analyses of simulations using sophisticated statistical techniques such as Bayesian statistics, cluster analysis, etc.

Acknowledgement

The author thanks Professor Wayne Getz and Dr. Andy Lyon of the University of California at Berkeley for their comments and contributions to this paper.

References

[1] P. A. Fishwick (Ed.), Handbook of Dynamic Sysgtem Modeling, Chapman & Hall, 2007, p. xiii.
[2] B. Richmond, A User's Guide to STELLA, High Performance Systems, 1985.
[3] R. Eberlein, D. Peterson, Understanding models with vensim¢, European journal of operational research 59 (1) (1992) 216–219.
[4] R. Macey, G. Oster, T. Zahnley, Berkeley madonna users guide, University of California, Berkeley, CA.
[5] S. Tisue, U. Wilensky, Netlogo: A simple environment for modeling complexity, in: in International Conference on Complex Systems, 2004, pp. 16–21.
[6] J. W. Forrester, Industrial Dynamics, MIT Press, 1961.
[7] H. Abelson, et al., Turtle geometry: The computer as a medium for exploring mathematics, MIT press, 1986.
[8] D. Hanselman, B. Littlefield, Mastering MATLAB 5: A comprehensive tutorial and reference, Prentice Hall PTR, 1997.
[9] R. Team, et al., R: A language and environment for statistical computing, R Foundation Statistical Computing.
[10] A. M. Starfield, R. M. Salter, Thoughts on a general undergraduate modeling course and software to support it, Transactions of the Royal Society of South Africa 65 (2).
[11] J. Dabney, T. Harman, Mastering Simulink 4, Prentice Hall PTR, 2001.
[12] P. Fritzson, Principles of object-oriented modeling and simulation with Modelica 2.1, Vol. 830938647, IEEE press, 2004.
[13] D. Flanagan, JavaScript: the definitive guide, O'Reilly Media, Incorporated, 2006.
[14] M. Gardner, The game of life, Scientific American 223 (1970) 120–123.
[15] U. Wilensky, Netlogo flocking model, `http://ccl.northwestern.edu/netlogo/models/Flocking`, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. (1998).
[16] C. Reynolds, Flocks, herds and schools: A distributed behavioral model, in: ACM SIGGRAPH Computer Graphics, Vol. 21, ACM, 1987, pp. 25–34.
[17] U. Wilensky, Netlogo ants model, `http://ccl.northwestern.edu/netlogo/models/Ants`, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. (1997).
[18] T. Ormsby, E. Napoleon, R. Burke, Getting to Know ArcGIS Desktop: The Basics of ArcView, ArcEditor, and ArcInfo Updated for ArcGIS 9, Esri Press, 2004.
[19] M. Neteler, H. Mitasova, Open source GIS: a GRASS GIS approach, Vol. 689, Kluwer Academic Pub, 2002.