An Introduction to *NovaScript*

Richard M. Salter

April 13, 2014

1 Introduction

NovaScript is a new language for defining simulations. *NovaScript* is embedded in and extends Javascript, and the *NovaScript* interpreter is an extension of the ECMA 1.7 Javascript standard. The full range of the Javascript language is available to the *NovaScript* programmer for specifying the computations required by his or her simulation.

NovaScript is part of the *Nova 2* simulation platform, which includes facilities for specifying simulations visually using an extension of the stock and flow diagrams familiar to programmers of Stella and Vensim. In *Nova 2* simulations may be specified visually using such visual schematic diagrams, or textually using *NovaScript*. We will refer to the former as a *diagram* specification, while the latter will be referred to as a *code* specification. *Nova 2* currently provides a mechanism for converting from diagram to code and future versions will provide an analogous tool for converting from code to diagram.

One major advantage of *Nova 2* and *NovaScript* is their ability to express and carry out integrated simulations defined in the system dynamics, cellular and agent-based paradigms.

2 System Dynamics Fundamentals

The reader is expected to be familiar with the basic structure of popular stock-and-flow-based simulation systems such as Stella, Vensim and Berkeley Madonna.

NovaScript is designed as an object-based system. Each object can have one or more fields containing values or expressions to be computed. Object behavior is determined by methods (functions) that define operations.

2.1 Primitive Components

Simulations in *NovaScript* are built out of objects called **components**. Among the component types are **primitive components**, **simulators**, **displays**, and **controls**.

The primitive components consist of **stocks**, **variables**, **sequences**, **flows** and **terms**. Stocks, variables and sequences are used to represent the state of a running simulation. The stock component may be attached to one or more flows. Each flow specifies some change in the current value of the stock. A variable is similar, only it is not attached to any flow. Its change is determined by a derivative contained in the *prime* field. Sequences are similar to variables, except that they are discrete, and change is indicated by a *next* field, directly determining the value of the sequence in the next time interval.

Flows and terms both have *exp* fields containing their expressions. As mentioned above, flows affect the value on the stocks to which they are attached. A term is used to compute a value using current state values (in Stella, what we call a term is referred to as a *converter*.)

2.2 Capsules

The **capsule** is one of the four types of **simulators** currently supported by *NovaScript* (the others are **cellmatrix**, **agentvector** and **simworld**; they will be discussed in subsequent sections). A simulator is a component capable of undertaking a complete simulation. Of the four, the capsule is the simplest. The role of the capsule is to contain the components of a simulation and synchronize the progress of the computation. A capsule holds instances of the primitive components, but may also contain other capsules, and other simulators.

2.2.1 Defining a capsule

A **level-one simulation** is defined by specifying a capsule. A capsule can be defined either directly or using a schema. In the example that follows we will be introducing and describing Javascript notation to the extent that it is required to understand *NovaScript*. The reader is encouraged to explore Javascript more deeply. One good tutorial is found at `http://www.cs.brown.edu/courses/bridge/1998/res/javascript/javascript-tutorial.html`. Note, however, that Javascript has been most popular as a Web-browser tool, which can be ignored for our purposes (you can start with Chapter 3)

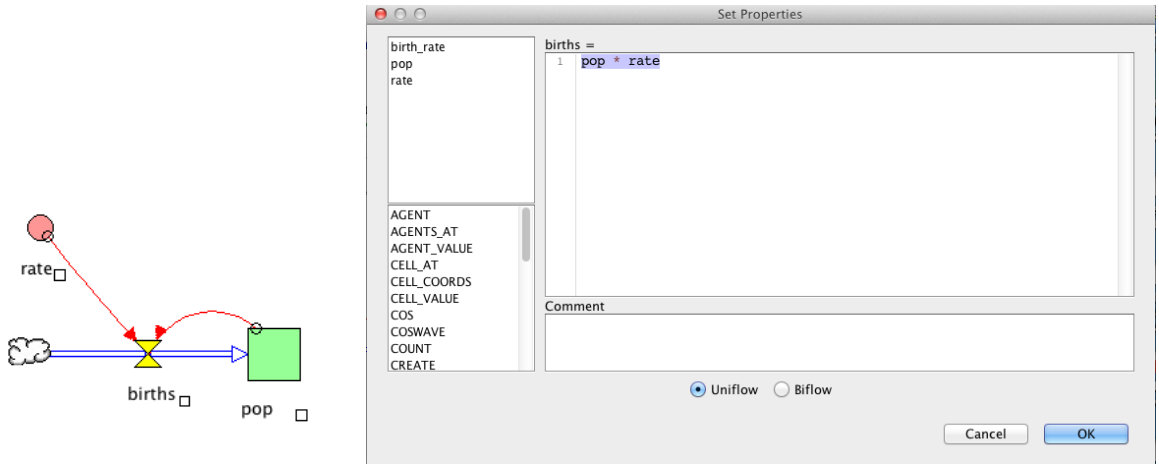

Figure 1: **Nova Population Model**

2.2.2 A simple example

Figure **??** is a *Nova 2* diagram depicting a simple model of exponential population growth. In Figure **??**b we have specified the expression (`pop * rate`) in the flow `births` governing the growth of the population represented by the stock `pop`. The value of `rate` (not shown) is a constant 0.1.

One direct representation of this scenario in *NovaScript* is as follows:

```
var PopCapsuleBuilder = function (name, start, r) {
    var pop = new Stock("pop");
    var births = new Flow("births");
    var rate = new Term("rate");
    pop.initial = start;
    births.exp = 'rate * pop';
    births.output = pop;
    rate.exp = r;
    pop.verbose = true;
    var ans = new Capsule(name, [pop, rate, births]);
    return ans;
};
```

We will now examine the features of the Javascript language being employed in this definition. Greater detail can be found in the Javascript tutorial referenced above.

`var PopCapsuleBuilder`

The `var` tag indicates a Javascript variable, to which some Javascript value can be assigned. Javascript values include numbers, strings, arrays, objects and functions.

`function (name, start, r) { ... }`

In this case we've assigned a function. Much of Javascript programming involves the definition of functions, which are computations that can be invoked multiple times. Here we are defining a function that will build a capsule according to our wishes.

`var pop = new Stock("pop")`

The right hand side indicates the creation of a new Stock object with name "pop". The term `Stock` in `new Stock(...)` is called a *constructor* since it builds a stock object.

All *NovaScript* components are given names. It is convenient, though not necessary, to assign this new object to a variable of the same name. Variables `births` and `rate` are being respectively assigned a flow and term.

`pop.initial = start`

The `initial` field of the stock `pop` is assigned the value of `start`, which is a parameter passed to the function. A field is a variable name associated with a particular object. The notation `pop.initial` is used throughout Javascript to reference fields. The `initial` field of a stock, variable or sequence is its initial value. It can be a constant or expression.

`births.exp = 'rate * pop'`

Here we assign the flow `births` the expression `rate * pop`. The expression takes the form of a string – a sequence of characters enclosed within single or double quotes (note that we use single quotes here but used double quotes when naming "pop", "births", and "rate"; the choice of single or double quotes is only important when one has one quoted string inside of another.) Strings are one of the two basic data types represented in Javascript (the other is numbers).

`births.output = pop`

With this assignment we are connecting the flow `births` to the stock `pop`. The reference to `pop` in this expression is the variable representing the stock of the same name.

`rate.exp = r`

We are setting the expression field of the term `rate` with the parameter `r` passed to the function.

`pop.verbose = true`

This causes the value of `pop` to be logged on the console as the simulation runs. We will see the effect later.

`var ans = new Capsule(name, [pop, rate, births])`

Having built each of the components, we now construct the capsule by passing a name and

list of components. The notation `[pop, rate, births]` is an array, or list. In Javascript an array can built out of any set of constituants;, e.g., `var a = [1, "two", 3.0]`. Array elements are accessed using square brackets and indices;, e.g. `a[0] == 1, a[1] == "two"`, etc.

`return ans`

The variable `ans` refers to the capsule we have just built; it is returned as the value of the function `PopCapsuleBuilder`.

Note that with this program we have created not just a single capsule, but a capsule "factory" that will produce as many instances of the population model as we'd like. By varying the parameters passed to the function, we obtain population models with different rates and starting values.

2.3 The Clock

Before we can run the example we need a special object called a *clock* Clocks are not simulation components; rather they initialize and control the running of the simulation. To create a clock you do something like the following:

```
var cl = new Clock(0, 20, .1, 'rk4')
```

The four parameters passed to the clock constructor represent the following:

start The starting time of the simulation.

end The ending time of the simulation.

dt The value of DT used to increment the current time at each step; also used by the integrator to determine state values.

method The integration method to be used. Acceptable values are *rk2*, *rk4* and *euler*.

2.4 Running the example

Enter the code from the listing above into the top window of the *NovaScript* console, along with the line defining the clock, and click **Load Program**. Next type the following into the *NovaScript* console lower window and click return.

```
js> var popModel = PopCapsuleBuilder("popModel", 1, 0.1);
```

You have defined `popModel` to be a population model capsule with start value 1 and rate 0.1.

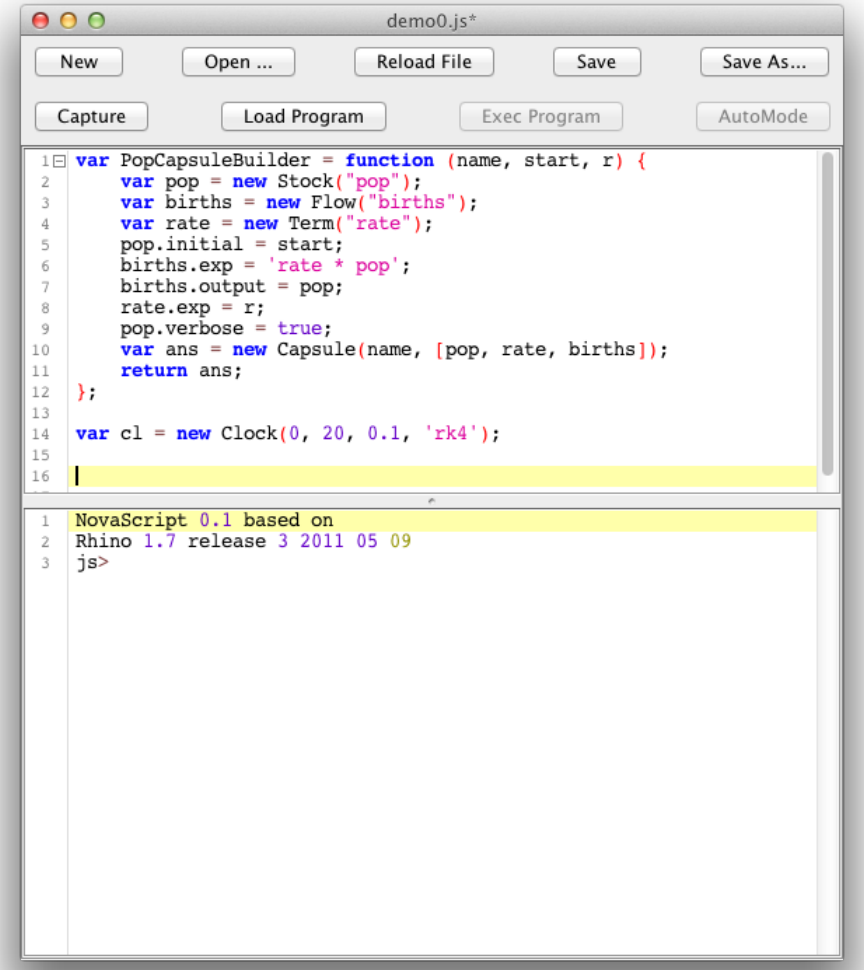

Figure 2: **Population Model in NovaScript**

To run the program, do the following (lower window):

```
js> cl.init(popModel)
js> cl.run()
```

Because of the `pop.verbose = true;` entry in our definition of `PopCapsuleBuilder`, we see the values of `pop` between 0 and 20 at intervals of 0.1.

Note: the single expression

```
js> cl.exec(popModel)
```

may be used as an alternative to

```
js> cl.init(popModel)
js> cl.run()
```

2.5 Alternate construct

Because this model uses a single flow into its only stock, we can eliminate the flow and substitute a variable for that stock. This version is defined as follows:

```
var PopCapsuleBuilder = function (name, start, r) {
    var pop = new Variable("pop");
    var rate = new Term("rate");
    pop.initial = start;
    pop.prime = 'rate * pop';
    rate.exp = r;
    pop.verbose = true;
    var ans = new Capsule(name, [pop, rate]);
    return ans;
};
```

3 Schemas

In the previous example, we built the capsule by constructing all of the component parts, connecting them directly through assignments to their fields, and passing the list to the capsule constructor. An alternate approach is to use *schemas*, which are structures that contain descriptions of the components and define their relationships. *NovaScript* has built-in tools that can create capsules (in fact all simulators, displays and controls) from these descriptions.

Schemas use a Javascript construct that we have not encountered previously: the Javascript object, also known as a record structure. A Javascript record structure is similar to an array in that it contains component element; however each element is associated with a *field*. The field is used both to assign and retrieve the data from the structure.

The form of a property list is shown in the following example:

```
var record = {first_name: "John", last_name: "Public", town: "AnyTown, USA"}
```

Here, the property names are `first_name, last_name` and `town`. Corresponding to `first_name` is the string `"John"`, etc. *Note: it is important that colons be used in associating fields with their values, and that commas be used to separate the field/value pairs.*

References to a record field can use either of two notations: the dot notation: `record.first_name`; and the square bracket notation: `record["first_name"]` both will return `"John''`. (Note the use of quotes in the second version).

Here is a schema defining the population model:

```
var popSchema = {
    specifies: "Capsule",
    properties: {r : 0.1},
    variables: 'pop',
    terms: ['birth', 'rate'],
    dynamics:[
        'pop.initial = 1',
        'birth = rate * pop',
        'pop.prime = birth',
        'rate = r',
        'pop.verbose = true',
    ]
}
```

The schema consists of a record containing the fields `specifies`, `properties`, `variables`, `terms` and `dynamics`. The order in which the fields appear is not meaningful.

The field value associated with `specifies` gives the type of component being defined by the schema; it includes all of the simulator types (`capsule`, `agentvector`, `cellmatrix`, `simworld`), as well as displays (`graph`, `table`, `raster`, `agentviewerx`) and controls (`slider`, `spinner`). The `properties` field is itself a record structure defining various constant values used in the expressions that follow below.

Next come the various component types. In this scenario we have listed a single variable, (named `pop`) and 2 terms (`birth` and `rate`). The right hand side of each of these fields must either be a string (if there is only one being declared) or a list of strings.

The largest field by far is `dynamics`. This consists of list of strings, where each string is an equation defining some property of some component. Note that these strings very closely match the body of our first definition.

To build a capsule with a schema, use the following form:

```
var popModel = Capsule.newCapsule("popModel", popSchema);
```

The capsule `popDemo` can be executed using a clock as described above.

3.1 Schema Registries

In the previous example we assigned a schema to the variable `popSchema`, then passed that variable to `Capsule.newCapsule` . An alternative approach is to associate a schema with a name using `defineSchema`, then pass the name rather than the variable when building the capsule. For example, our population model schema can be associated with the name `"popSchema"` as follows:

```
defineSchema("popSchema", {
    specifies: "Capsule",
    properties: {r : 0.1},
    variables: 'pop',
    terms: ['birth', 'rate'],
    dynamics:[
        'pop.initial = 1',
        'birth = rate * pop',
        'pop.prime = birth',
        'rate = r',
        'pop.verbose = true',
    ]
});
```

Now, creating a capsule looks like this:

```
var popModel = Capsule.newCapsule("popModel", "popSchema");
```

Almost the same, but note the quotes around `popSchema` – we pass the name rather than the structure. *NovaScript* maintains a *schema registry* where `defineSchema` associates names with

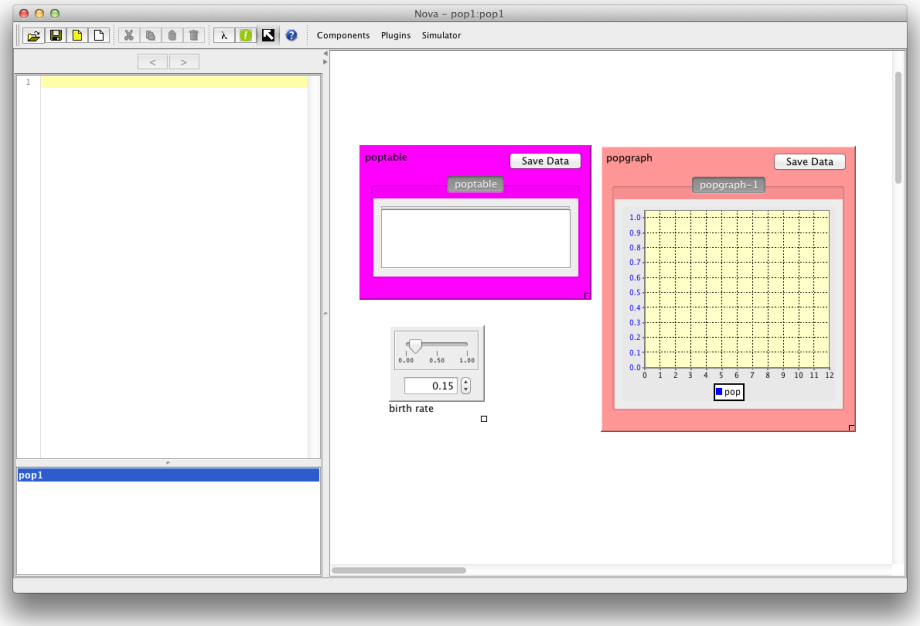

Figure 3: **Nova Display and Control Components**

schema structures. The advantage of this approach is one of efficiency: a registered schema has a single capsule factory associated with it to build all of the capsules required by a particular model.

4 Displays and Controls

The *Nova 2* visual platform contains several visualizing components (tables, graphs) and control components such as sliders and spinners. *NovaScript* programs that don't originate visually can still use these to present data and specify input values.

Figure **??** shows the *Nova 2* platform containing a table, graph and slider. These can be used in a coded program by adding "proxy schemas" to that program as follows:

```
pop1.defineSchema('my_slider', {
    specifies: "Slider",
    proxy: "birth_rate",
    lo: 0.000,
    hi: 1.000,
    dec: 2,
    value: 0.150
});
```

```
pop1.defineSchema('my_table', {
    specifies: "Table",
    proxy: "poptable",
    display: "pop"
});

pop1.defineSchema(my_graph', {
    specifies: "Graph",
    proxy: "popgraph",
    type: "TIMESERIES",
    display: "pop"
});
```

A proxy schema represents a visual display or control within a coded program. [Note: the names chosen for the schema are arbitrary]. The `proxy` field must match the name of the visual element to make the connection. The other fields provide configuration information for the control or display. In the case of the table and graph, the `display` field indicates what is being tabulated or graphed.

Finally, we add 3 components (`birth_rate`, `poptable` and `popgraph`) to our model definition and associate them with the proxy schemas. Since the display field of the 2 display schemas indicate what is to be shown, no further elaboration is required. We do, however, have to connect the slider with the `rate` term; this can be seen in the new equation `'rate = birth_rate'`. We have also deleted the property `r` since it is no longer being used.

```
defineSchema("popSchema", {
    specifies: "Capsule",
    variables: 'pop',
    terms: ['birth', 'rate'],
    controls : {
        birth_rate: "my_slider",
    },
    displays: {
        poptable: "my_table",
        popgraph: "my_graph",
    },
    dynamics:[
        'pop.initial = 1',
        'birth = rate * pop',
        'pop.prime = birth',
```

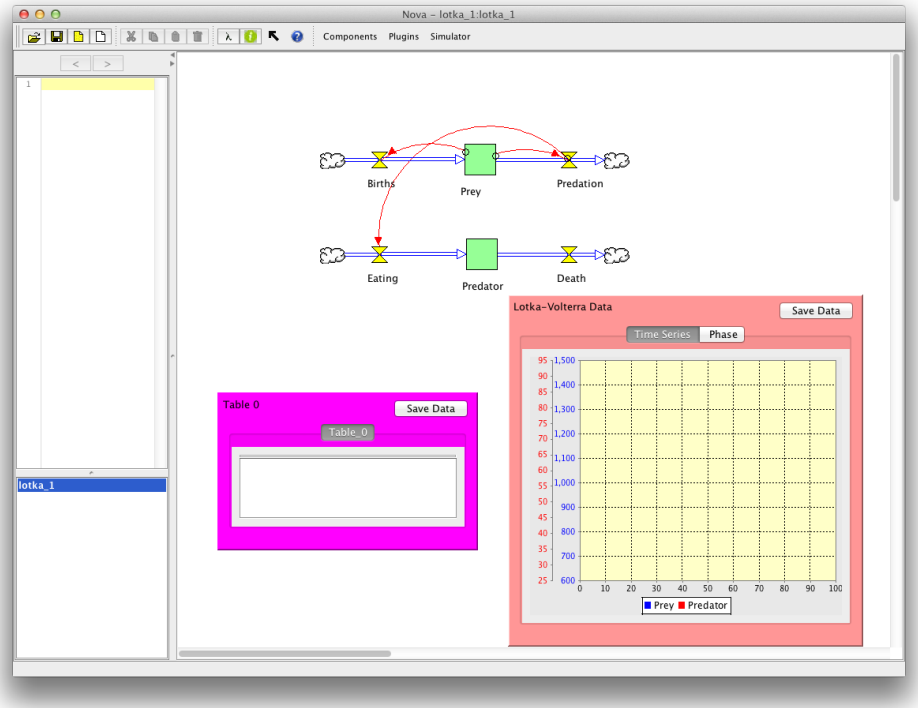

Figure 4: **Lotka-Volterra Model**

```
        'rate = birth_rate',
        'pop.verbose = true',
    ]
});
```

5 Project-Based Models

NovaScript allows you to organize all aspects of a given simulation into a *project*. Projects are particularly important for the capture mechanism, which creates a project as the target of the capture.

To illustrate, we consider the diagrammed implementation of the Lotka-Volterra model shown in Figure **??**. A coded implementation of this model will require 3 scenarios (one for the capsule and one for each of the displays). Clicking **capture** on this model produces the following code:

```
var lotka_1 = new Project('lotka_1');
beginProject(lotka_1);

lotka_1.defineSchema('main', {
```

```
    specifies: "CAPSULE",
    stocks: ['Prey', 'Predator'],
    flows: ['Eating', 'Death', 'Births', 'Predation'],
    displays: {
        Table: "main_Table",
        Lotka-Volterra_Data: "main_Lotka-Volterra_Data",
    },
    dynamics: [
        "Prey.initial = Init_Prey",
        "Prey.nonnegative = true",
        "Prey.output = Predation",
        "Predator.initial = Init_Pred",
        "Predator.nonnegative = true",
        "Predator.output = Death",
        "Eating.output = Predator",
        "Eating = c * Predation",
        "Death = m * Predator",
        "Births.output = Prey",
        "Births = r * Prey",
        "Predation = (g * Prey) * Predator",
    ],
})

// Displays & Controls

lotka_1.defineSchema('main_Table', {
    specifies: "Table",
    proxy: "Table",
    display: "[[Predator, Prey]]"
});
lotka_1.defineSchema('main_Lotka-Volterra_Data', {
    specifies: "Graph",
    proxy: "Lotka-Volterra Data",
    type: "[TIMESERIES, SCATTER]",
    connectDots: "[false, true]",
    display: "[[Prey, Predator], [Prey, Predator]]"
});

// Aux
...
```

```
endProject(0.000, 100.000, 0.500, 'rk4');
```

The first line creates the project. Next come bracketing commands `beginProject` and `endProject`. Note that the defineSchema commands are methods of the project rather than global functions. [Note: we are ignoring the `Aux` section since it only contains information concerning the appearance and settings of the visual components]. The parameters passed to `endProject` create a clock for this project, stored in the variable `$clock$`.

5.1 Running a Project

Projects can be run by clicking the `Load Program`/`Exec Program` buttons. They can also be run in the console by typing

```
js> init();
js> run();
```

or simply

```
js> exec();
```