Getting started with the .NET version of MODCOM

Frits van Evert and Ben Rutgers (frits.vanevert, ben.rutgers)@wur.nl

21 September 2007 (original version: 18 October 2005)
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>2</td>
</tr>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Get the software</td>
<td>3</td>
</tr>
<tr>
<td>A brief overview of modeling and simulation</td>
<td>4</td>
</tr>
<tr>
<td>Implement a model using MODCOM</td>
<td>6</td>
</tr>
<tr>
<td>Run a simulation with one component model</td>
<td>11</td>
</tr>
<tr>
<td>Run a simulation with two component models</td>
<td>12</td>
</tr>
<tr>
<td>SimBuilder: specify and run simulations</td>
<td>14</td>
</tr>
<tr>
<td>Using <code>mrun.exe</code> to run simulations</td>
<td>15</td>
</tr>
<tr>
<td>Compiling MODCOM components with MONO</td>
<td>16</td>
</tr>
<tr>
<td>A closer look at the MODCOM framework</td>
<td>17</td>
</tr>
<tr>
<td>Class diagram</td>
<td>17</td>
</tr>
<tr>
<td>Time events and the event list</td>
<td>17</td>
</tr>
<tr>
<td>State events</td>
<td>18</td>
</tr>
<tr>
<td>How integration is handled in MODCOM</td>
<td>19</td>
</tr>
<tr>
<td>Time</td>
<td>19</td>
</tr>
<tr>
<td>Examples with relevance to biophysical simulations</td>
<td>19</td>
</tr>
<tr>
<td>Crop/weed/soil using one resource</td>
<td>19</td>
</tr>
<tr>
<td>Farm management</td>
<td>19</td>
</tr>
<tr>
<td>MODCOM source code repository</td>
<td>19</td>
</tr>
<tr>
<td>Further reading</td>
<td>20</td>
</tr>
<tr>
<td>Glossary</td>
<td>21</td>
</tr>
<tr>
<td>Literature cited</td>
<td>22</td>
</tr>
</tbody>
</table>
Introduction

This manual describes:

- How to write component models for the MODCOM framework
- How to create and run simulations using component models that were implemented by yourself or by others.

The organization of this manual is as follows. First, the context is defined in which implementing models and running simulations takes place. Then, detailed instructions are given about how to implement component models. Next, the steps needed to create and run simulations are given. Several topics that are touched upon briefly in the first sections are discussed in more depth in the last part of this manual.

Get the software

Download the file nmodcom.zip from http://www.modcom.wur.nl/. Open the file, extract NModcom.dll and store it somewhere on your computer.
A brief overview of modeling and simulation

A system is the part of reality that we consider in a particular study. Thus, a system is defined by its boundaries (Fig. 1).

Figure 1. A system defined by its boundaries.

Over time, a system may be in different states. A system consisting of a ball in a classroom is the same system regardless of the position and velocity of that ball, but it will be in a different state when the ball is lying on the floor than when the ball is at a height of 1 m above the floor and falling. An important aspect in constructing a model of a system is determining its state variables, that is, identifying those quantities that must be given a value to completely describe the state of the system.

The transition of the system from one state to another is described by the change in value for each state variable. A quantitative model of the system is a set of equations that can be used to calculate the change in value for each state variable. These equations use the current state of the system as well as extraneous inputs.

Differential equations are a common method to describe transitions in biophysical systems (Fig. 2).

\[
\begin{align*}
\frac{dX_1}{dt} &= f(X_1, X_2, \ldots, X_n) \\
\frac{dX_2}{dt} &= f(X_1, X_2, \ldots, X_n) \\
& \vdots \\
\frac{dX_n}{dt} &= f(X_1, X_2, \ldots, X_n)
\end{align*}
\]

Figure 2. System defined by its boundaries and by a set of differential equations.

Given a set of different equations that describes state transitions in the system of interest, we can write software that implements the model represented by the equations. For small systems, this is simple and straightforward. The pseudo-code of a computer program that implements a model with two state variables is shown in Fig. 3.

```
for (time = startTime; time < endTime; time = time + timeStep)
{
    // calculate rates
    rX1 = ... (code that calculates the rate of change of state variable X1)
    rX2 = ... (idem for X2)
    // update states
    X1 = X1 + rX1 * timeStep;
    X2 = X2 + rX2 * timeStep;
}
```

Figure 3. Outline of a computer program that is capable of simulating a system with two states.
For large systems, there are at least three reasons that make it desirable to define sub-systems: 1) manage complexity at the conceptual level, 2) manage complexity at software level, and 3) re-use software components in several whole-system softwares.

Figure 4. System conceptualized as consisting of two sub-systems. Each sub-system is represented by a set of differential equations. Differential equations for one sub-system may be expressed partly in terms of the state of the other sub-system; thus, solving the equations representing one sub-system may require knowledge about the state of the other sub-system. This is expressed by the double-headed arrow connecting the two sub-systems.

Given the above decomposition of the original system into two sub-systems, we can implement the model as two distinct pieces of software (Fig. 5).

```c
for (time = startTime; time < endTime; time = time + timeStep) {
    // calculate rates for the first subsystem
    // inputs are X1 and X2
    // output is rX1
    rX1 = submodelA.rates(X2)

    // calculate rates for the second subsystem
    // inputs are X1 and X2
    // output is rX2
    rX2 = submodelB.rates(X1)

    // update states
    X1 = X1 + rX1 * timeStep;
    X2 = X2 + rX2 * timeStep;
}
```

Figure 5. Outline of a computer program that is capable of simulating a system with two states. The rate calculations take place in software components called “submodelA” and “submodelB”. Here, these software components are merely linked; they have been developed separately.

MODCOM is a framework that allows the linking of model implementations. MODCOM handles communication between component models, numerical integration, and events handling.
Implement a model using MODCOM

In this section we will implement a model component to be used in MODCOM. We will use the version of MODCOM that was written for Microsoft’s .NET. We will use the computer language C# to implement the model; and we will use Microsoft’s Visual Studio 2003 as our development environment. It is possible to use tools from other vendors. In a subsequent section we will use Mono to compile and run our models.

1. Start Microsoft Visual Studio .NET 2003 by selecting
   Start
   Programs
   Microsoft Visual Studio .NET 2003
   Microsoft Visual Studio .NET 2003

The start up screen will appear:

![Start up screen](image)

2. From the start up screen choose New Project; the New Project dialog will appear, select Visual C# projects, Console Application; fill in an appropriate Name and Location for the new project:
3. The next screen will appear (this is the Development Environment for the C# application):

3. Our project should know where to find NModcom.dll; to do so follow these steps:
3.1. In the Solution Explorer window right click References and choose Add Reference…. When the Add Reference dialog appears, choose Browse....
3.3. The Select Component dialog appears; browse to the folder where you previously saved your copy of NModcom.dll, select the file and choose Open.

3.4. Back in the Add Reference dialog make sure that NModcom.dll is selected in the Selected Components: list and choose OK.

NModcom.dll has now been added to your project and is listed under “References” in the Solution Explorer on the right-hand side of the screen.

We can now start writing a model implementation.

4. Immediately after the line using System; add the line using NModcom; this makes that the classes defined in NModcom.dll can be used from our code.
4. Insert a new public class after the curly brace } of Class1. Give this class an appropriate name (e.g. MyFirstModel). Let it derive from OdeSimObj. This class implements a lot of housekeeping code and by using it as the starting point of our own code we gain access to its functionality without writing a line of code. The code should now look like this:

```csharp
using System;
using NModcom;

namespace MyModel
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            // The main entry point for the application.
            // [STAThread]
            start void Main([string] [] arg)
            {
                // TODO: Add code to start application here
            }
        }
    }
    public class MyFirstModel: OdeSimObj
    {
    }
}
```

5. We will implement a model of the growth of a population of one species in the presence of another species. This model has one state variable (“density”), two parameters (“rgr” is relative growth rate of our population; “k” denotes the influence of the density of the other species on the growth of our population), and one signal (“otherDensity” is the population density of the other species). Declare private fields for the state, the parameters and the signal; use type “double” for all of them:

```csharp
public class MyFirstModel: OdeSimObj
{
```
private double density;
private double rgr;
private double k;
private double densityOther;
}

6. Use [State()], [Param()] and [(Signal())] attributes on the above fields to state the meaning of the fields that you have just created. The first parameter to each attribute is a name that will be visible outside the component and will be used when linking your component to others; and the second parameter can be used to provide a link to an ontology that describes precisely and unambiguously the meaning of the field. An initial value for the variable can be provided as shown below.

```csharp
public class MyFirstModel: OdeSimObj
{
    [State("density", "?")]
    private double density = 3;
    [Param("rgr", "?")]
    private double rgr = 1;
    [Param("k", "?")]
    private double k = -0.5;
    [Signal("density of other species", "?")]
    private double densityOther;
}
```

7. Override the virtual GetRates() method and write your model equations. Store the rate of the first state variable in deriv[index], the rate of the second in deriv[index+1], and so on:

```csharp
public class MyFirstModel: OdeSimObj
{
    [State("density", "?")]
    private double density = 3;
    [Param("rgr", "?")]
    private double rgr = 1;
    [Param("k", "?")]
    private double k = -0.5;
    [Signal("density of other species", "?")]
    private double densityOther;

    public override void GetRates(double[] deriv, int index, double time)
    {
        //calculate rate
        deriv[index] = rgr * density + k * density * densityOther;
    }
}
```

8. Your new class is now complete. Test your code by selecting from the menu
Build, Build Solution.
If your code is OK you will receive the following message

```
Build: 1 succeeded, 0 failed, 0 skipped
```

If you receive build errors correct the code. Probably you have misspelled the name of a variable or keyword. Be aware of the fact that C# is case sensitive!

The implementation of your model is now finished.
Run a simulation with one component model

In the body of the main class's `static void Main()` method, add the lines that are shown below. The first line creates an instance of MODCOM's default simulation environment. The second line creates an instance of the model that you have just implemented. The third line places this instance under control of the simulation environment. The statement “`simenv.Run()`” in the fourth line instructs the simulation environment to run the simulation. Running a simulation means that initial values are assigned to state variables, current time is set to `simenv.StartTime`, and then integration steps are made until the current time is larger than or equal to `simenv.StopTime`. In each timestep, first `GetState()` is called so the simulation environment knows the current state of your module; then `GetDerivatives()` is called to get rate of change; next new states are calculated; and finally `SetState()` is called to store the new state in your module. This is with the Euler or rectangular integration method; other integrators, e.g. Runge-Kutta may make many calls to `Get/SetState()` and `GetDerivatives()` in one timestep.

class Class1
{
    [STAThread]
    static void Main(string[] args)
    {
        ISimEnv simenv = new SimEnv();
        ISimObj simobj = new MyFirstModel();
        simenv.Add(simobj);
        simenv.Run();
        Console.WriteLine("Simulation finished");
    }
}

Test your application following these steps:

- Place a breakpoint on the first line of the `Main()` method: place the cursor on the line and press F9, or click in the grey vertical bar to the left of the code area;
- Start the application using F5 or choose Debug, Start from the menu;
- Program flow will stop in the line with the breakpoint;
- Step through the code using F10.

Hint: if you put a breakpoint in the method `GetRates()` of your model implementation, you can see that the framework does call on your model and that it does so each time with different arguments.

The default values that we have specified (initial value of density = 3, rgr = 1, k = -0.5 and otherDensity = 0) result in unlimited exponential growth. Moreover, MODCOM uses by default an integration timestep of 1. This results in a simulated timeseries of population density of 3, 6, 12, 24, etc. You can change the timestep of the integration by adding the following statement somewhere before `simenv.Run()`:

`simenv.Integrator.IntegrationTimeStep = 0.01;`
Run a simulation with two component models

Replace the code in Main() with the lines shown below. This is very similar to what we have seen in the first section, except that now we have two instances of MyFirstModel, and they have suggestive names.

```csharp
static void Main(string[] args)
{
    SimEnv simenv = new SimEnv();
simenv.Integrator.IntegrationTimeStep = 0.01;
    ISimObj prey = new MyFirstModel();
simenv.Add(prey);
    ISimObj pred = new MyFirstModel();
simenv.Add(pred);
    simenv.Run();
}
```

Set predator values.

```csharp
static void Main(string[] args)
{
    SimEnv simenv = new SimEnv();
simenv.Integrator.IntegrationTimeStep = 0.01;
    ISimObj prey = new MyFirstModel();
simenv.Add(prey);
    ISimObj pred = new MyFirstModel();
simenv.Add(pred);
    pred.Inputs["rgr"].Data = new ConstFloatSimData(-1);
pred.Inputs["k"].Data = new ConstFloatSimData(0.5);
simenv.Run();
}
```

Connect.

```csharp
static void Main(string[] args)
{
    SimEnv simenv = new SimEnv();
simenv.Integrator.IntegrationTimeStep = 0.01;
    ISimObj prey = new MyFirstModel();
simenv.Add(prey);
    ISimObj pred = new MyFirstModel();
simenv.Add(pred);
    pred.Inputs["rgr"].Data = new ConstFloatSimData(-1);
pred.Inputs["k"].Data = new ConstFloatSimData(0.5);
pred.Inputs["Density of other species"].Data = prey.Outputs["Density"].Data;
pred.Inputs["Density of other species"].Data = pred.Individuals["Density"].Data;
simenv.Run();
}
```

Add the line “using NModcom.Util” at the top of the file; add an output writer and run the simulation. Output is now in file out.txt.

```csharp
static void Main(string[] args)
{
    SimEnv simenv = new SimEnv();
simenv.Integrator.IntegrationTimeStep = 0.01;
    ISimObj prey = new MyFirstModel();
simenv.Add(prey);
    ISimObj pred = new MyFirstModel();
simenv.Add(pred);
    pred.Inputs["rgr"].Data = new ConstFloatSimData(-1);
pred.Inputs["k"].Data = new ConstFloatSimData(0.5);
pred.Inputs["Density of other species"].Data = prey.Outputs["Density"].Data;
pred.Inputs["Density of other species"].Data = pred.Outputs["Density"].Data;
    OutputWriter outputter = new OutputWriter();
    outputter.TimeStep = 0.5;
}
```
This program creates several objects in the memory of your computer and establishes links between them. A schematic overview of these links (minus the OutputWriter) is given in Fig. x.

**Figure x.** Layout of computer memory while the simulation program in this section is being run. Objects are represented by circles. “Prey”, “Predator”, and “Euler” implement ISimObj; “SimEnv” implements ISimEnv. Objects without names represent unattached input numbers. Input ports of ISimObj’s are represented by rectangles; output ports are represented by small circles attached by a thick line. Relationships between objects are as follows: the SimEnv object “contains” three SimObj’s; the Euler object “integrates” both Prey and Predator; the various input ports “find data” either from an unattached input, or from the output port of another SimObj.
SimBuilder: specify and run simulations

SimBuilder is a GUI application that can be used to configure simulations. When the application is started, it scans all dll’s in its start-up directory (the directory where simbuilder.exe is located) for classes that implement the MODCOM ISimObj interface. Any classes found are listed in the list box on the left hand side of the screen and can be used to assemble simulations.

**Fig x.**

**Add a SimObj to the simulation:** click on the name in the list box, then click on the drawing canvas. Right-click to stop adding objects. SimObj’s are represented by a box with the name of the object listed at the top, input ports listed on the left, and output ports listed on the right.

**Connect two SimObj’s:** click on the arrow icon in the toolbar, click on an output port, move to an input port, click again.
**Using mrun.exe to run simulations**

SimBuilder simulations are stored in xml-files with the following format.

```xml
<?xml version="1.0"?>
<simenv starttime="0" stoptime="30">
  <simobj name="Predator1"
    class="NModcom.Examples.Predator" assembly="NModcom.dll" protocol=".NET"/>
  <simobj name="Prey1"
    class="NModcom.Examples.Prey" assembly="NModcom.dll" protocol=".NET"/>
  <simobjlink sourcenode="Predator1" destinationnode="Prey1">
    <simdatalink output="Density" input="Density of other species"/>
  </simobjlink>
  <simobjlink sourcenode="Prey1" destinationnode="Predator1">
    <simdatalink output="Density" input="Density of other species"/>
  </simobjlink>
</simenv>
```

These files can be run with the command-line application `mrun.exe` as follows:

```
mrun <myfile.sim>
```

where `<myfile.sim>` is the name of a file that was produced either by SimBuilder or by using a text editor.
Compiling MODCOM components with MONO

Mono is an open-source project that has produced a C# compiler and a runtime environment (http://www.go-mono.com). You can use Mono to compile your MODCOM components and run your simulations. Components compiled with Mono can use components compiled with Visual Studio, and components compiled with Visual Studio can use components compiled with Mono. Similarly, programs compiled with Mono can run on Microsoft’s .NET runtime, while programs compiled with Visual Studio can run on the Mono runtime. Finally, Mono is available for a number of Unix flavours, including, of course, Linux.

In this section we will use Mono to compile the source code from the previous section and run the program.

To compile your new application with Mono follow these steps:

- First copy NModcom.dll to the directory where you saved your C# application; to test the application you’ve just made in Visual Studio this directory would be “C:\usr\mary\MyModel” (according to the names used in this example).

- Start a Mono Command Prompt window:

  Start
  Programs
  Mono 1.1.9.2 for Windows
  Mono-1.1.9.2 Command Prompt

Make the directory where you stored your new application the active directory (using the DOS-command “cd”).

From the command prompt enter the following command:

  mcs /r:nmodcom.dll /t:exe /out:myapp.exe Class1.cs

If your code is correct the file myapp.exe will be made. If there are errors they will be reported. You can correct your code in any texteditor and compile it again.

You can run the program myapp.exe from the command prompt.
**A closer look at the MODCOM framework**

**Class diagram**
The UML class diagram in Fig. x. shows the most important classes in the MODCOM framework.

![Class diagram of MODCOM framework](image.png)

**Figure x.**
ISimObj
ISimEnv
IData
IIntegrator
ISimEvent

**Time events and the event list**
A MODCOM simulation is driven by time events. Time events are scheduled to occur at a certain time; at that time, the simulation object that is the target of the event is notified. Simulation objects respond to events in a way that makes sense to them. For example, integrator objects (objects that implement IIntegrator) perform an integration step.

Events are implemented as classes and thus can encapsulate arbitrary amounts of information. This makes it possible that a crop object responds to “Harvest” events, for example by reducing its biomass; and that a soil object responds to “Tillage” events.

The event list is not fixed: during a simulation new events can be added, while scheduled events can be removed. Thus, an integration object may schedule additional integration events if the time step of numerical integration must be changed; and a farm management object may schedule irrigation events as they become necessary.
The following code snippet shows how to create a time event and add it to the event list.

```java
TimeEvent e = new TimeEvent(this, target, time);
simenv.RegisterEvent(e);
```

**State events**

State events are similar in name, but quite different in function, to time events. Imagine that the temperature in a greenhouse is rising slowly. When the temperature reaches a certain threshold, windows are opened to slow or prevent a further rise in temperature. This means that as soon as the temperature reaches the threshold, the state of the system must be changed (windows opened). We do not know at which point in time the threshold will be reached. It is most likely that there will be an integration step at the beginning of which the threshold is not yet reached, while at the end of the step the threshold has been overshot. In order to simulate this situation correctly, we must iteratively change the last integration step until we reach precisely the point in time at which the state event occurs; change the state of the system; and continue the simulation. In Fig. x, the integration step denoted by arrow 1 would result in an overshoot. Halving the time step still results in an overshoot (arrow 2). Halving the time step again results in undershoot (arrow 3). In this example, the final time step will be somewhere between arrows 2 and 3.

Creating and registering a state event is a bit more complicated than creating and registering a time event, because a state event must be hooked up with an algorithm to determine whether the condition it must detect has occurred.
How integration is handled in MODCOM

The UML sequence diagram in Fig. x shows the method calls that are made during one integration step, using the Euler integrator.

![UML sequence diagram](image)

Figure x.

Time
How to deal with dates, year-of-rotation, etc.

Examples with relevance to biophysical simulations

Crop/weed/soil using one resource

Farm management

MODCOM source code repository

The source code for MODCOM is available at [http://www.modcom.wur.nl/](http://www.modcom.wur.nl/).
Further reading

A good textbook on modeling and simulation of ecological processes is Leffelaar (1993). A general theory of modeling and simulation (and a framework that looks a lot like MODCOM) is given by Zeigler et al. (2000). Finally, Hillyer et al. (2003) describe an earlier version of MODCOM.
<table>
<thead>
<tr>
<th><strong>Glossary</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
</tr>
<tr>
<td><strong>Component model</strong></td>
</tr>
<tr>
<td><strong>Framework</strong></td>
</tr>
<tr>
<td><strong>Instance</strong></td>
</tr>
<tr>
<td><strong>Interface</strong></td>
</tr>
<tr>
<td><strong>Model</strong></td>
</tr>
<tr>
<td><strong>Object</strong></td>
</tr>
<tr>
<td><strong>Simulation</strong></td>
</tr>
<tr>
<td><strong>Sub-system</strong></td>
</tr>
<tr>
<td><strong>System</strong></td>
</tr>
</tbody>
</table>
Literature cited


Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. Design Patterns: elements of reusable object-oriented software Addison-Wesley.

