The ModCom modular simulation system

Charles Hillyer a,*, John Bolte a,1, Frits van Evert b,2, Arjan Lamaker c,3

a Department of Bioengineering, Gilmore Hall, Oregon State University, Corvallis, OR 97331, USA
b Plant Research International, P.O. Box 16, 6700 AA Wageningen, The Netherlands
c IMAG, Mansholtlaan 10-12, 6708 PA Wageningen, The Netherlands

Abstract

Simulation models of agro-ecological systems are typically written in a manner that precludes reusability of parts of the model without a significant amount of familiarity with and rewriting of existing code. Similarly, replacing a part of a model with a functionally equivalent part from another model is typically difficult. The objective of this study was to develop a method to enable the assembly of simulation models from previously and independently developed component models. Recent advances in software engineering have enabled the development of software applications from smaller parts (called components) on the basis of an abstract decomposition of the relevant domain (called a framework). Based on a requirements analysis of existing simulation models we developed the ModCom simulation framework. ModCom provides a set of interface specifications that describe components in a simulation. ModCom also provides implementations of the core simulation services. The framework interfaces use well-defined binary standards and allows developers to implement the interfaces using a broad range of computer languages. Using this framework, simulation models can be assembled by connecting component models in much the same way that Lego blocks are put together to assemble a house. ModCom thus allows modelers to create models and modeling tools that are easily exchanged (in binary form or source code) with colleagues across the hall or across the globe.

#2002 Elsevier Science B.V. All rights reserved.

Keywords: Modular simulation; Simulation frameworks; Object oriented; Agro-ecological simulation

1. Introduction

Mathematical systems models and computer simulations are used by ecological scientists and resource managers to enhance ecosystem management, allocate resources, and understand ecological and biological processes. Simulation models become vehicles for representing in abstract terms a slice of reality consistent with our specific interests in enhancing our understanding of our world. Representation here is the key word: simulation models must provide us with an
abstract representation of reality. How we represent the conceptual model we derive from observation of reality therefore should be of considerable interest to the modeler. It is here where programming languages and simulation tools have a central role to play.

In the formulation and description of these models, it is both useful and common to think in terms of models and sub-models. Scientists in ecology will readily recognize the usefulness of considering, for example, a sub-model for crop growth and a sub-model for nitrogen transformation and movement in the soil, in the case of an agricultural model. Yet the computer representations of these models often show little sign of these conceptual decompositions. If an attempt is made to have the structure of a computer program resemble the structure of the conceptual model, for example by using subroutines, the parts of such a computer program are often still linked tightly together and cannot be replaced by equivalent parts from programs developed elsewhere.

Object-oriented techniques and, in particular component-based software development hold promise to obtain a high degree of correspondence between conceptual models and their implementation in computer code. This has led researchers to develop object-based models in a variety of domains. Numerous object-oriented models have been reported in ecological and agricultural simulations (Sequeria et al., 1991; Whittaker et al., 1991; Van Evert and Campbell, 1994; Folse et al., 1990; Caldwell and Fernandez, 1998; Bolte, 1998). All indicated support for model conceptualization, program design, and reuse of the models as advantages of object-oriented approaches for model development.

The objective of this research is to develop a method that enables the assembly of simulation models from previously and independently developed models. The following sections describe the technique (frameworks) and technology (component software) that are used to achieve the objective. Following that is a description of the ModCom simulation framework and its associated parts.

2. Frameworks

An object-oriented framework is a set of collaborating classes meant to be expanded to form related applications (Gamma et al., 1995). A simulation framework, then, is a set of classes meant to be expanded to create simulation programs. Such a framework may describe the flow of execution, patterns of communication, or data structures used by elements of a simulation. The purpose of constructing a framework is twofold. First, the framework creates a separation of concerns by segregating the application-specific parts of simulations from the application-independent code employed by many simulations to accomplish common tasks. Creating this separation greatly enhances code reuse. The second purpose is to create a clear path for building a simulation. By defining what elements of the framework actually contain the model’s implementation and how those elements are used, a designer is presented with a clear path from conceptual model to simulation. Further, by defining how model components to interact with other components, a simulation framework greatly enhances the reusability of model components, allows the development of robust metamodeling facilities (e.g. parameter estimators, stochastic analysis capabilities), and speeds the assembly and analysis of complex models. Frequently used classes are provided by the framework, removing the need for individual modelers to ‘reinvent the wheel.’

A framework is manifested as a collection of interfaces (abstract classes), together with concrete classes that implement frequently needed functionality. An interface is an agreement between developers defining the semantics of how to communicate with an object implementing the interface. Programmatically, an interface defines a collection of related methods that implement the interface semantics. If a class implements some interface then it must implement that interface fully and exactly according to the definition of the interface. Thus, when a programmer uses an object implementing the interface, he or she knows what methods the object has, what the method arguments are, and has general knowledge of what the
object will do when one of it’s methods is invoked. Practically speaking, an interface is an abstract class containing only method definitions without implementations. The interface does not specify how those methods should be implemented. This separation of implementation from definition is an essential part of achieving language independence. The separation also facilitates the management of changes to the code after it has been deployed.

There are drawbacks to using a framework. The designers of a framework aim to support certain type of applications (Gamma et al., 1995). The development of applications is facilitated by the framework as long as they are of this type; if the applications are sufficiently different from what the framework designers had in mind, the framework ceases to be useful.

An example of a high-level decision support framework was developed by Bolte et al. (1993) and Bolte (1998). This framework provided for the integration of continuous, event-driven, and knowledge-based simulations by providing two major classes and several supporting classes. The first of these, termed the Simulation Environment, provides for a number of different types of simulation components, including continuous simulators, discrete events, and knowledge-based agents, each of which could be subclassed into more specific types of simulation objects. The simulation environment provides a simulation clock controlling and coordinating time-based operations, maintains an event list of ‘interesting’ events scheduled to occur at some point in future time, provides a number of different notification and message-passing mechanisms allowing communication and interaction between objects at several different levels via messages directed to specific objects, general notification messages, and a blackboard supporting asynchronous communication between objects in the system. Because all objects in the system are derived from a single high-level simulation class, all user-defined simulation components automatically receive robust simulation capabilities, and integration of conventional continuous simulators running at variable time steps, periodic and aperiodic discrete events and expert system-based agents is straightforward.

Other modular simulation frameworks have been developed. Loki, a modular system for X Windows, has been used successfully for ecosystem modeling and forest fire management (Keane et al., 1996). The USGS developed the Modular Modeling System (Leavesley et al., 1996) to address problems of model selection and application for environmental and water resource problems. The High Level Architecture, a general-purpose architecture for simulation reuse and interoperability, was developed by the Defense Modeling and Simulation Office of the Department of Defense (Dahamann et al., 1998). The Modular Modeling Language employs a meta-model approach to module construction and has been used to develop spatial ecosystem models (Maxwell and Costanza, 1997). While not strictly object-oriented, these systems have demonstrated in part the utility of frameworks for modular model development. However, these approaches have had problems with language dependence, lack of robust communication and identification capabilities among modules in the system, and lack of robust time and information flow sequence coordination between modules.

Agro-ecological simulations typically present numerous requirements which are common across systems, including (1) standardized public interfaces defining object access and action initiation, (2) high-level communications capabilities for components of the system to communicate with other components in a non-specific manner, (3) standardized methodologies for collecting and transferring information between components of the system, possibly in a networked or web-based environment, (4) standardized methods for data import, representation, analysis, visualization and export, and (5) mechanisms for synchronizing the sequencing of flow execution among system components. The framework paradigm provides potentially useful capabilities in all of these areas. The standardization of interfaces is a central framework concept, and is readily implemented through the definition of generic interfaces that provide a consistent specification for how objects in a simulation interact with each other and with the framework. An standardized interface specification allows communication between objects
without specific knowledge of object’s implementation, a critical requirement for the development of high-level, domain-independent modeling frameworks. Synchronization and data collection can similarly be handled in a standardized manner through the specification of high-level interfaces.

3. ModCom

ModCom is a framework for developing and using modular simulation components. The framework is supplied with a library (ModComLib) that provides components that implement many of the interfaces defined in the framework and provide access to core simulation services. Additionally, development tools supporting rapid creation of components and visual model assembly are being developed.

We wanted ModCom to have the following characteristics: First, the system must be practical and easy to use. The system must be capable of exploiting existing protocols for object communication and data sharing, but this complexity should be hidden from the user as much as possible. A visual tool should be available to help automate the process of model construction and execution.

Several design goals were set for the development of ModCom. First, ModCom should be language neutral. To achieve language and operating system independence, ModCom uses the Component Object Model (COM), an industry standard binary specification of interface definitions. While COM is both a specification and a set of platform-specific libraries, ModCom uses only the COM specification to maintain platform-independence. A second goal was that the entire system should be extensible: a developer should be able to replace any framework component with a different implementation. Finally, modules should be independent of each other. A developer should be able to construct a module without incurring runtime dependencies between other modules.

4. Core simulation management services

The elements of the ModCom framework are grouped according to the services that they provide. Each service is made available as a set of interfaces that provide access to an implementation of the service. The following sections describe the core simulation services that are used by all simulations. Fig. 1 shows a class diagram of the framework with the interfaces grouped according to the services that they provide. Here, we provide a high-level description of the ModCom framework. Specific details about methods are available in the ModCom technical documentation. Interested readers should examine the ModCom technical documentation and users guide for additional details (see Section 13).

5. Simulation objects

A ModCom simulation consists of individual, interacting components implementing one or more ModCom interfaces. Each of the components in a simulation exposes inputs and outputs and communicates with other objects in the system. For example, a crop model might consist of several simulation components: a plant component that implements a series of state equations describing dynamic plant response, a climate-generation component, a soil water balance component, a data writing component periodically collecting and writing simulation results to a file or database, and, if a visual representation of results is needed, a dynamic graphing component that periodically collects results and displays them on a screen. Different components (e.g. a different climate generator) can be quickly swapped into an assembled model as long as they implement at least the ISimObj interface. These interacting components are called simulation objects (or SimObjs). Each SimObj must implement one or more of the following interfaces, but at a minimum, components in a simulation must implement the ISimObj interface described below (*NOTE*: interface names are prefixed with a capital ‘I’).
5.1. The IsimObj interface

The ISimObj interface specifies basic object identification and data exposure methods and must be implemented by all objects that participate in a simulation. However, many simulation components have more specialized requirements. For example, many model components are represented as a system of differential equation based state variables. Because all objects of this type will require numerical integration services to be solved, additional interfaces are defined that allow general-purpose integrators to solve these object’s state equations without the modeler having to implement these methods. But, at a minimum, all objects must implement the ISimObj interface. The object identification is simply a Name parameter exposed as a string. The data exposure methods are more complicated and are discussed in the section on Exposable Data and the ISimData interface.

5.2. The IUpdateable interface

The IUpdateable interface allows a SimObj to be updateable; that is, it will receive periodic messages from a SimEnv (described below) to ‘update’ itself through the SimEnv’s time flow synchronization mechanism. Many SimObjs will implement IUpdateable as well as ISimObj.

Objects can interpret what it means to update themselves in their own context. An integrator might update itself by solving the SimObjs associated with it by integrating their state equations for one time step. A graph might update itself by refreshing its data store and redrawing itself. Update messages are provided by the SimEnv based on the updateable SimObj’s time step. Updateable SimObjs can control when they start and stop receiving messages, their time step, when they receive their next update, and other aspects of updating.

Fig. 1. UML class diagram of the core ModCom interface. Interfaces have been grouped according to the services they provide. Inheritance means that one class ‘inherits’ functionality from its parent class. Aggregation means that one class contains a reference to another class.
5.3. The IODEProvider interface

The IODEProvider interface defines support for SimObjs that are represented by one or more ordinary differential equations (ODE). The methods defined by IODEProvider allow integrator objects (described later) to solve these differential equations in a generalized way without requiring the modeler to specify solution procedures for the equations they are implementing.

6. Simulation management

6.1. The simulation environment

ModCom provides support for high-level management of simulations through the ISimEnv interfaces. The interface defines functionality in the following areas:

(1) Registration and management of simulation objects participating in a simulation. Registration is a process where the simulation environment is made aware of simulation objects that want to participate in a simulation. Upon registration the SimEnv determines what messages the SimObj will receive based on what interfaces it implements and schedules the SimObj for the appropriate services.

(2) Time-flow synchronization and control of the updateable SimObjs. A SimEnv accomplishes time-flow synchronization by maintaining an event list that schedules the execution of the IUpdateable.Update method according to each Updateable’s TimeStep and priority information.

(3) Default integration methods for numerical solution of ODE. The SimEnv implementation that is part of ModCom has access to an implementation of IIntegrator (described in Section 7). The SimEnv will schedule SimObjs for integration services; however, the Integrator actually performs the integration. Additional or alternative integrators can be substituted for the default Integrator.

(4) Broker services for inter-object communication. The SimEnv maintains a store of SimData and SimDataInfo (described in Section 9). Other objects may query the SimEnv for specific variables (SimData) or variables with particular set of attributes.

(5) Initializing, stopping, and other execution control functions. The SimEnv is intended to be a simulation controller. It has methods to set the simulation start time and stop time, and to run a simulation.

6.2. Simulation events

In addition to simulation objects, the ModCom framework supports the concept of a simulation event. An event is simply something that gets executed at a specific point in simulated time. Events have no state; that is, they do not maintain any data. Internally, the SimEnv handles updating simulation objects through an update event scheduler, but modelers can also define their own events and register them with the SimEnv to perform event-driven tasks. Within the framework, events are defined using the ISimEvent interface. The framework provides several general-purpose implementations of this interface. As with all interfaces, modelers can create additional event classes by implementing the ISimEvent interface.

7. Numerical integration services

Numerical integration for ODEs is accomplished through an implementation of the IIntegrator interface. Objects designed to provide integration services must implement the ISimObj and IUpdateable interfaces in addition to IIntegrator. The object that provides integration services should perform its calculations during the call to IUpdateable.Update. These two requirements allow the integrator to interact with the SimEnv as a normal updateable SimObj thus simplifying the SimEnv and allowing multiple integrators to exist simultaneously.

An integrator implementation accomplishes integration by maintaining a list of all the ODEProvider objects that require integration services. At each time step the integrator’s IUpdateable.Update method is called. At that point the integrator should collect each ODEProviders state and deri-
data to be exchanged between SimObjs via a series of ISimObj methods. The framework provides a default implementation for ISimData that should satisfy most data exchange needs. Hence individual components should not need to provide an implementation of this interface.

The ISimObj interface defines methods to allow a SimObj to expose any data it wants to make public. This information can be an internal SimObj variable or a derived variable resulting from a computation. The SimData are accessible to other SimObjs via the ISimObj.Output property. As the property name implies SimData exposed via the Output property are intended to be used by other SimObjs, not written to. The ISimObj interface also has an Input property that allows a SimObj to define what information the SimObj itself will use. This combination of Inputs and Outputs defines a rudimentary asynchronous data flow model for combining modules.

The data stored in a SimData is exposed on the interface (and stored internally) as a VARIANT type. The VARIANT data type is a Microsoft standard for containing both fundamental data types such as strings or floating-point variables, and abstract data types in a language neutral manner. As such, a broad range of types can be exchanged between objects using SimData. ISimData defines methods for accessing the VARIANT directly, or as a scalar type (i.e. integer, floating point or string). In addition to providing data, a SimData can provide descriptive information about itself through the ISimInputInfo and ISimOutputInfo interfaces. These two interfaces provide information about the SimData such as data type, physical units, and runtime behavior.

Exposing a variable (one that the module wishes to make available to other components) involves the following steps. First, a SimData is instantiate for each datum that will be exposed. Second, the SimData is made available to other SimObjs (and the SimEnv) via the ISimObj.Output property. Finally, the ISimData.Value property of the SimData instance is maintained by the SimObj throughout the SimObj’s lifetime. A detailed example of this procedure is available in the ModCom users guide (see Section 13).

### Table 1
Integration methods available with the default IIntegrator implementation

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euler</td>
<td>Simple Euler finite difference method</td>
</tr>
<tr>
<td>RK2</td>
<td>Second Order Runge–Kutta method</td>
</tr>
<tr>
<td>RK4</td>
<td>Fourth Order Runge–Kutta method</td>
</tr>
<tr>
<td>RKF</td>
<td>Fifth order adaptive Runge–Kutta–Feldberg method</td>
</tr>
<tr>
<td>RKCK</td>
<td>Fifth order adaptive Runge–Kutta–Feldberg method using Kash–Karp coefficients</td>
</tr>
</tbody>
</table>

8. Exposable data and the ISimData interface

An important capability of any module-based framework is the ability for different components in the framework to be able to communicate and pass information between each other. The framework defines the ISimData interface for allowing...
9. Connection protocols for inter-object communication

The ISimEnv interface is responsible for maintaining a store of SimData references; it serves as a ‘broker’ of the data. In addition to storing these variables, the SimEnv automatically builds data reference collections based on the SimObjs exposure of SimData, and allows for querying for specific variables by other objects in the system.

Importing an output from one component to use in another can occur in one of two ways. In a loosely coupled system, where each component has no knowledge of the other a SimObj can query the SimEnv, prior to conducting a simulation. The queried SimData can then be assigned to a SimObjs Input property and can used as a source of information during a simulation. This process of coupling one SimObjs output to another’s input can be performed by the objects themselves or by some object acting outside the SimEnv (e.g. a simulation tool).

In a more tightly coupled system, SimObjs can request information (specific SimData) from other components that the SimObj knows about without querying the SimEnv. This method allows greater efficiency of data exchange. The second approach should be used cautiously, as one of the advantages of the framework is the ability to loosely couple objects. In either case, the queries for SimData will return a reference to the SimData exported by another component; the querying object should store this reference (as an input SimData) and refer to it as needed through the ISimData interface methods.

10. Data management services

One of the core services provided by the framework is data management. Virtually all simulations involve the reading, manipulation, sharing, and writing of data. The framework provides a standard method for defining data flow and representation to facilitate data-related simulation tasks. These services are provided through the use of four interfaces: IRowColDataset, ICollectOutput, IDataReader, and IDataWriter.

10.1. IRowColDataset

Because ‘rectangular’ datasets, tabular data arranged in rows and columns, are ubiquitous in simulations, support for such datasets is provided by the framework through definition of and implementation of the IRowColDataset. This interface provides methods for accessing, creating, and managing a rectangular dataset based on the VARIANT data type.

10.2. IDataReader and IDataWriter

The IDataReader and IDataWriter are intended to provide access to stores of IRowColDatasets. Both interfaces provide a means to specify the source of the data, cause the data to be loaded or stored, and the means to access the data as an instance of IRowColDataset. It is assumed that a class that implements IDataReader or IDataWriter will provide access to a specific type of database. For example, one DataReader may provide access to tables through the Microsoft ADO API while another might use specially formatted text files. The purpose behind including IDataReader and IDataWriter is to allow developers to leverage existing code bases for reading and writing specialized data files.

10.3. ICollectOutput

Implementation of ICollectOutput provides an object participating in a simulation the opportunity to record its output during a simulation. The interface also exposes the recorded data as an IRowColDataset allowing interoperability with the IDataReader and IDataWriter interfaces. The rate at which output collection occurs is specified by the SimObj independently of the update rate of the SimObj. Scheduling of output collection is handled automatically when the SimObj is registered with the SimEnv.

11. Using ModCom

ModCom can be used to assemble and execute a simulation by connecting ModCom-compliant
components. If the required components are not available, a developer can create them by implementing specific ModCom interfaces with a COM enabled development environment such as Visual Basic, Visual C++, or Delphi. When creating a SimObj, each component must at least implement ISimObj. SimObjs whose inputs and outputs will change regularly during a simulation should implement IUpdateable. Objects that represent ODE’s should implement IODEProvider.

There are some design issues that should be weighed when building a SimObj. The ModCom framework defines SimObjs with the intent that they will be combined with other SimObjs to assemble a simulation. The SimObjs are, logically, parts of (or sub-models of) some larger model. When creating a SimObj the designer should keep in mind that the SimObj will be used as a part of a simulation rather than a simulation by itself.

SimObjs built with ModCom can be distributed in binary form as well as in the traditional form of source code. The SimObjs, once registered on a user’s computer, are available in any COM enabled development environment (e.g. Excel, Visual Basic, Delphi, etc.). Building and running a simulation with ModCom involves five basic steps. An example simulation involving a predator/prey system is shown in Fig. 2.

```vba
Sub Macro1()

' Example Predator Prey simulation

'Step 1: create the SimObjs and SimEnv
Dim env As New SimEnv
Dim predator As New Predator
Dim prey As New Prey
Dim preyData As RowColDataset
Dim predData As RowColDataset
Dim i, rows As Integer

'Step 2: register the SimObjs
env.Register prey
env.Register predator

'Step 3: connect the SimObjs
prey.Input(0) = predator.Output(0)
predator.Input(0) = prey.Output(0)

'Step 4: run the simulation
env.StartTime = StartTime 'variables copied from the worksheet
env.StopTime = StopTime
env.Run

'Step 5: copy data into the worksheet
Dim output As ICollectOutput
Set output = prey
Set preyData = output.DataObject
Set output = predator
Set predData = output.DataObject

rows = predData.rows
For i = 0 To rows - 1
   Worksheets(1).Cells(i + 1, 1).Value = predData.Get(i, 1)
   Worksheets(1).Cells(i + 1, 2).Value = preyData.Get(i, 1)
Next i

End Sub
```

Fig. 2. Predator/Prey example written with Excel VBA.
In this example (written in Excel VBA) the SimObjs, predator and prey, both implement a single differential equation. When solved together they represent a simple Lotka–Volterra type predator prey system. The input and output values exposed via the SimObj interface represent the predator and prey densities. While this program may be of little practical value, it demonstrates the potential of modular simulation. First, the details of integration management are hidden in the SimEnv. Second, suppose the author wanted to consider the effect of a more complex predator. Using a different SimObj to replace the existing predator would only require that the example code be changed (in fact, only the declaration). The SimEnv, and the existing prey SimObj would not require any modification. Furthermore, a new predator could be written in a different language from the existing prey or the example program.

Because of space limitations we cannot show the implementations of the predator and prey classes. Microsoft COM code tends to be especially verbose however much of it is generated automatically by COM enabled development environments. A complete implementation is available at the ModCom website (see Section 13) as well as a users guide that provides a detailed description of how to build a SimObj.

There is an additional method of using ModCom that involves no programming at all. One of the design goals for ModCom was to enable visual model assembly tools. An additional interface, ISimObjView, was defined for this purpose. The ISimObjView interface allows SimObjs to display themselves and provide display information to visual design tools. One such tool being developed to use the ModCom interfaces is the Visual Modeling Environment (VME). It allows users to connect SimObjs by manipulating graphical representations of the objects. Using VME users can assemble models from existing components without programming.

12. Conclusions

ModCom is a robust and versatile framework for developing and using agro-ecological simulations. The framework is language neutral; ModCom Simulation Objects can be developed in any language environment that supports COM. The framework is also fully extendable. Any component can be replaced with a different implementation without affecting the other components. Numerical integration services, time flow synchronization, and data exchange services all simplify the development of simulation modules and facilitate a ‘plug and play’ style of modular simulation.

At present all of the core simulation management services have been implemented. Rigorous testing has been performed and the ModCom library is stable. A test suite was developed concurrently with the library to help with debugging and as an exact expression of the libraries runtime specifications. The test suite covers all of the functionality provided by the ModCom library and is available with the ModCom distribution. Current activities involve two areas of development. First, we are working with several other groups to develop a set of modules that will be of practical use to agro-ecological modelers. The second area of development involves further development of the ModCom framework. In particular we are developing tools that will automate parts of the module construction process. These tools will integrate with several development environments (e.g. Visual C++, Visual Basic, Delphi, etc.) and will be provided as part of the ModCom distribution.

13. Availability of software

The ModCom source code, and associated development materials are available on the web at http://biosys.bre.orst.edu/modcom.

Acknowledgements

This work was supported in part by EPA grant # R82-7960.
References