The Society for engineering in agricultural, food, and biological systems

An ASAE Meeting Presentation

Paper Number: 053011

Development of a Forage Growth Component in the Object Modeling System

Allan A. Andales[1], Soil Scientist

[1]USDA-ARS-Great Plains Systems Research Unit, 2150 Centre Ave., Bldg. D, Suite 200, Fort Collins, CO 80526, Allan.Andales@ars.usda.gov

Olaf David, Research Scientist

Colorado State University, Dept. of Civil Engineering, Fort Collins, CO 80523, Olaf.David@ars.usda.gov

Lajpat R. Ahuja[1], Soil Scientist / Research Leader

Laj.Ahuja@ars.usda.gov

Written for presentation at the
2005 ASAE Annual International Meeting
Sponsored by ASAE
Tampa Convention Center
Tampa, Florida
17 - 20 July 2005

Abstract. *The Object Modeling System (OMS) is a framework that facilitates the development of customized models from a standardized library of science, data and utility modules, as well as their testing, application and deployment. It is an interagency project between the USDA-ARS, USGS, and USDA-NRCS to implement object-oriented modeling principles that achieve code reusability and greater ease of maintenance. This paper demonstrates the development of a rangeland forage component in OMS using two approaches: (1) wrapping of an existing non-object-oriented forage module written in FORTRAN 90; and (2) creating a fully object-oriented forage module written in Java. The first approach demonstrates reuse of legacy code from an existing model while the second approach demonstrates the development of a Java OMS component. Features of OMS relevant to component development are also highlighted. OMS can leverage previous investments in legacy science modules and facilitate the development of component-oriented models within a framework that maximizes code reusability and ease of maintenance.*

Keywords. Object Modeling System, OMS, modeling framework, component oriented programming, modularity, forage growth

Introduction

Multidisciplinary environmental simulation models integrate approaches from a wide range of disciplines such as hydrology, soil science, plant physiology, ecology, and others. Usually, expertise in these fields can be found across research agencies, universities and other institutions. Integrating a complex model from different parts becomes not only a scientific challenge but a technical implementation and social engineering challenge as well. That issue becomes more pressing under the current pace of model development with development cycles being reduced drastically. The conventional approach of creating a large simulation model in a monolithic structure is bound to fail for a modeling project that is interdisciplinary and requires contributions from distributed modeling groups.

The current trend in software engineering favors a short-term development process where large and complex applications are built using a series of smaller parts, referred to as components. Component technology has been the most sustainable step in the evolution of software design and development over the past years. It has been adopted for simulation model development for different modeling applications and frameworks (Argent, 2004).

What is a model component? It is a model application-level software unit that is ideally developed for a specific purpose and not for a single model application. Components in general are self-contained, large-grained software entities. They are modeling units that are context-independent both in the conceptual and technical domain. A modeling component usually addresses one modeling concept at a certain level of complexity. Some examples are a precipitation, erosion, or a plant growth component. Technically a modeling component represents an executable code which has enough information bundled with it to explain itself to model builder tools with regards to its purpose, scale, capabilities, structure, data requirements, and other model application-relevant properties.

The objective of this paper is to give an overview of components in the Object Modeling System (OMS; David et al., 2002) and demonstrate the development of a rangeland forage component in OMS using two approaches: (1) wrapping of an existing non-object-oriented forage module written in FORTRAN 90; and (2) creating a fully object-oriented forage module written in Java. The first approach demonstrates reuse of legacy code from an existing model while the second approach demonstrates the development of a Java OMS component.

The Object Modeling System

The Object Modeling System is computer software that helps to (i) create and test scientific simulation components, (ii) package components, (iii) assemble components into a model application, and (iv) apply a model against datasets and analyze model outputs graphically. OMS supports the management lifecycle for all modeling elements.

OMS utilizes Component Oriented Programming (COP) for simulation models (Ahuja et al., 2005). With respect to COP, modeling components are depicted by some main characteristics:

- OMS embraces the JavaBeans component standard. JavaBeans is the component architecture for the Java platform. It defines a standard for the structure of Java classes, which can be used as pluggable components in application builder tools.
- OMS components have meta data. Meta data is used to describe components with respect to their modeling domain. Component data is described in terms of data types, data units, constraints, and default values. Components, in general, have attached meta data for information about the component such as description, scales of application,

literature references, author, version, etc. For a more detailed discussion of meta data in the OMS, the reader is referred to David et al. (2004).

- OMS components are implemented in the Java programming language. However, OMS provides tools to create Java mediation components for existing simulation sources written in other languages such as FORTRAN. This feature helps integrate proven legacy code into OMS in a transparent way.

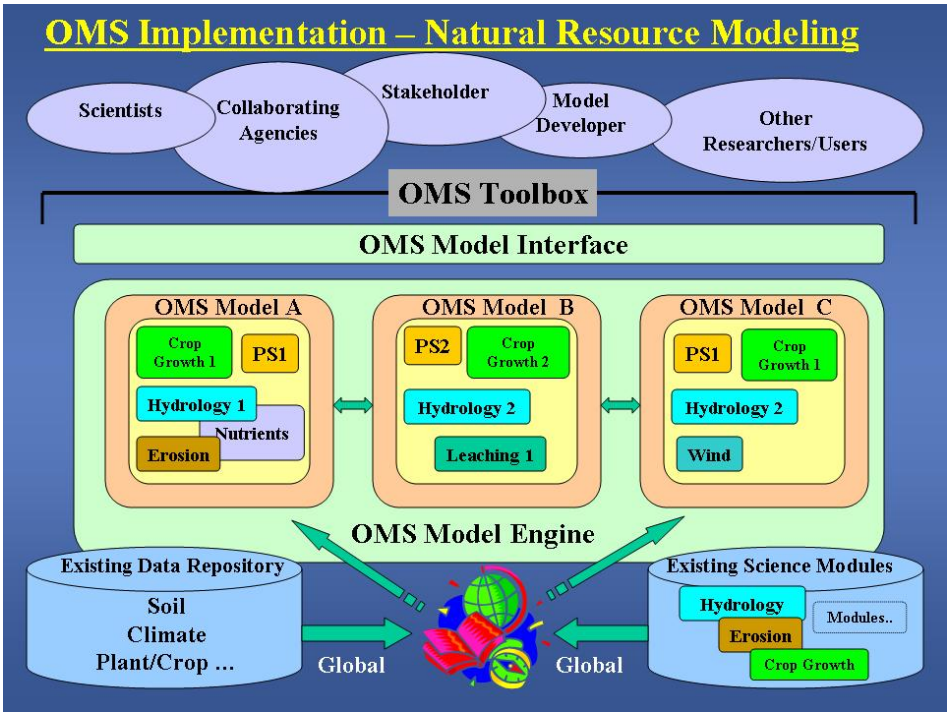

Figure 1. Schematic of OMS implementation for natural resource modeling.

Figure 1 shows the OMS implementation schematic for natural resource modeling currently implemented at the ARS Great Plaint System Research Unit, Fort Collins in collaboration with the NRCS Information Technology Center, Fort Collins. The building blocks for customized models are contained in a library of science, control, and database components (OMS components). There are currently several parallel efforts to build and integrate hydrology, erosion, range-livestock, nutrients, and other components for several research projects.

This paper focuses on the development of a forage growth component in OMS as an example. The reader is referred to David et al. (2002) and Ahuja et al. (2005) for more comprehensive descriptions of the OMS.

OMS Components

A component representing a certain conceptual function in a simulation is a building block for each model. A hydrological model for example usually needs components for handling input/output, has components representing processes of a hydrological system such as precipitation, interception and runoff, and has to implement general data-processing functions such as reading climate data or parameter sets. The key for a proper model design based on

components is a clean "separation of concerns" in a model. Components for a model hosted in OMS have to provide a certain aspect. What qualifies a part of a model as a component?

1. A component has a specific and usually single conceptual function in a model. It represents a physical process, a management action, a data gathering part, or the presentation of results to the user interface. Such functions need to be identified and separated from each other. Each of these aspects will result in a component.
2. An identified component can be fully described with regards to its function, data requirements and data offerings. Therefore, the specification and subsequent implementation of a component will be done with respect to its anticipated simulation context, but a tight dependency to this context is avoided. Later in the process of development, the component will be tested standalone using a test bed environment to prove its correct functioning.
3. The component is general enough to be used in other models and applications. So designing and implementing it will eventually require more work at the beginning, but will definitely pay off when it gets reused and re-purposed later.

By analyzing simulation models a classification of potential components can be made.

- **Scientific components** implement methods and equations to estimate some physical phenomenon. Examples would be a component estimating amount of water evaporated from a certain land cover, a component predicting the soil loss due to wind erosion, etc. Such components usually apply some mathematical function.
- **Scientific utility components** support the analysis of models by providing statistical analysis methods such as descriptive statistics, frequency analysis, etc. Distribution generator components are used to provide data to scientific components.
- **Control components** are responsible for managing the execution of a model. A Runge-Kutta Integration component, a time management, or a convergence criteria component are examples of this.
- **Data Input/Output Components** provide data to other components in a simulation model. Such components could handle data transfer from databases or files to the model. Visualization components like graphs or spreadsheets also fall under this category.

From the technical perspective the parts of a component in OMS may be classified into two main categories.

- **Component behavior** is specified by Java Interfaces. The OMS API offers several interface types that support different component types. There can be stateless and stateful components, visual components, or components which have an implementation in FORTRAN or C (native). A component may also have a combination of these features. All components have an execute() method that implements the desired functionality (e.g. simulation of a process). The execute() method may either call legacy code (e.g. FORTRAN, C++ dlls) using the Java Native Interface (JNI) (Kralisch et al., 2004) or call Java code. This facilitates the incorporation of pre-existing legacy code and leverages previous investments in scientific code. The components presented in

this paper demonstrate the native FORTRAN and the default Java stateful behavior.

- **Component structure** is specified by attributes. Attributes are properties from the JavaBeans perspective. Attributes mainly represent the data flow into and out of the component. They can also be seen as parameters or variables from the conventional modeling perspective. Attributes have meta data attached to them to indicate characteristics such as data flow, physical units, range constraints, or the intentional role this attribute is supposed to play in a model (variable or parameter). Each attribute is accessed via 'getter' and 'setter' functions that return and set its value, respectively.

Development of Forage Growth Component

The forage growth model that was converted into an OMS component was taken from the Great Plains Framework for Agricultural Resource Management (GPFARM) decision support system. Andales et al. (2005) give pertinent details of the model. Live and dead forage (above-ground biomass) are simulated for 5 functional groups: warm-season grasses, cool-season grasses, forage legumes, shrubs and unpalatable forbs. Daily forage production is simulated based on environmental inputs including daily maximum and minimum air temperature (°C) and a non-dimensional water stress factor (0 – 1) defined as the ratio of actual transpiration and potential transpiration.

Component Meta Data

The header of the component contains a brief description of the forage component and meta data documenting the author, version, and available reference (Fig. 2). Other optional meta data may be included, all of which are prefixed with '@oms'.

```
/**
 * Estimates daily shoot (above-ground) growth in rangelands.
 * @oms.author Allan Andales
 * @oms.version 1.0
 * @oms.reference Andales et al. 2005. Rangeland Ecology &  Management 58(3):247-255
 */

public class Forage implements Stateful, Native {
   // User code
}
```

Figure 2. Forage component meta data.

Native Component (Wrapping of Legacy Code)

The OMS is able to integrate legacy code modules. By an automated JAVA wrapper generation for legacy code, modules written in languages such as Fortran or C++ can be converted into OMS components at the function or subroutine level.

The existing forage growth model was written in FORTRAN 90 and included the following program units:

- Function Forage – calculates daily growth and senescence of each of the five functional groups;
- Module ForagePlants – contains the FORTRAN 90 type definition for a generic plant (mainly declarations of variables used to describe the state of a plant) and several generic growth response functions;
- Module SiteDep – contains the FORTRAN 90 type definitions for describing site-dependent characteristics (e.g. maximum forage production, area, proportion of the plant community in each functional group, etc.).

The function Forage "uses" the modules ForagePlants and SiteDep.

The Forage component is 'Stateful'. This means that it manages its own memory resources by implementing an init() method to initialize attributes (variables and parameters) with appropriate values, open files, or connect to external databases; and a cleanup() method to re-initialize values or possibly close open files or other connections to system resources (Fig. 3). The key feature of the native Forage component is that it implements the 'Native' interface, which indicates that the execute() method is implemented through a call to an external FORTRAN function called 'Forage' (Fig. 3). The external call to FORTRAN function Forage involves the passing of values between the OMS (ByteBuffer) and the external function through facilities of the Java Native Interface (JNI). Note the JNI meta data prefixed by '@jni'.

```
class Forage implements Stateful, Native {

   public void init() { // Initialization code }

   public void cleanup() { // Cleanup code }

   public void execute() {
      Forage(((BufferBacked)day).getBuffer(), ((BufferBacked)EaEp).getBuffer(),
                  ((BufferBacked)NitStress).getBuffer(), ((BufferBacked)AveHeight).getBuffer(), ...);
   }
/**
 * Implementation of 'Forage'
 * @jni.arg.native day I
 * @jni.arg.native EaEp D
 * @jni.arg.native NitStress D
 * @jni.arg.native AveHeight D ...
 */
   private native void Forage(ByteBuffer day, ByteBuffer EaEp, ByteBuffer NitStress,
                                    ByteBuffer AveHeight, ...);
}
```

Figure 3. OMS forage growth component (in Java) that calls legacy FORTRAN code. Ellipses (…) indicate code not shown.

Operationally, the Java code cannot call the FORTRAN function directly. Through the JNI, an intermediate C function (code not shown) that mirrors the passing arguments in Forage(…) is called and subsequently passes the argument values to the FORTRAN function. The intermediate C function serves as the bridge between the Java code and the legacy FORTRAN code. Thus, calling legacy FORTRAN code from the OMS involves the sequence: OMS component (Java) $\rightarrow$ C/C++ bridge $\rightarrow$ FORTRAN legacy code. The OMS handles this automatically and the process is completely transparent to the user.

All variables in the legacy code that need to be accessible to the OMS must be included in the subroutine or function argument list. This means that any variable contained in a FORTRAN common block, module, or type data object must be "brought out" into the function or subroutine argument list for the OMS to access it. In the case of the original FORTRAN forage code, essential variables contained in the ForagePlants and SiteDep modules were "brought out" and added to the argument list of function Forage.

Java Component

A Java OMS component is defined as one that is written entirely in Java (the language of the OMS), without calls to legacy code. Thus, the model can be fully object-oriented and all attributes and methods (e.g. process calculations) are contained in Java classes. As an example, the procedural (FORTRAN) and monolithic forage growth module described in the previous section was re-conceptualized (Fig. 4) and re-programmed in Java. Each box in Figure 4 represents an OMS component. A plant or functional group is modeled as having a phenology, shoot, and root component. A rangeland plant community is then modeled as an aggregate of several plants or functional groups. In this example, warm season grasses (WSG), cool season grasses (CSG), legumes, shrubs, and forbs are each modeled at the plant level and then aggregated to form a plant community. Pertinent details of the behavior of the OMS Shoot component are shown (Fig. 5) as an example of a Java component. The other components of the object-oriented forage growth model follow a similar pattern but are not shown here because of space limitations.

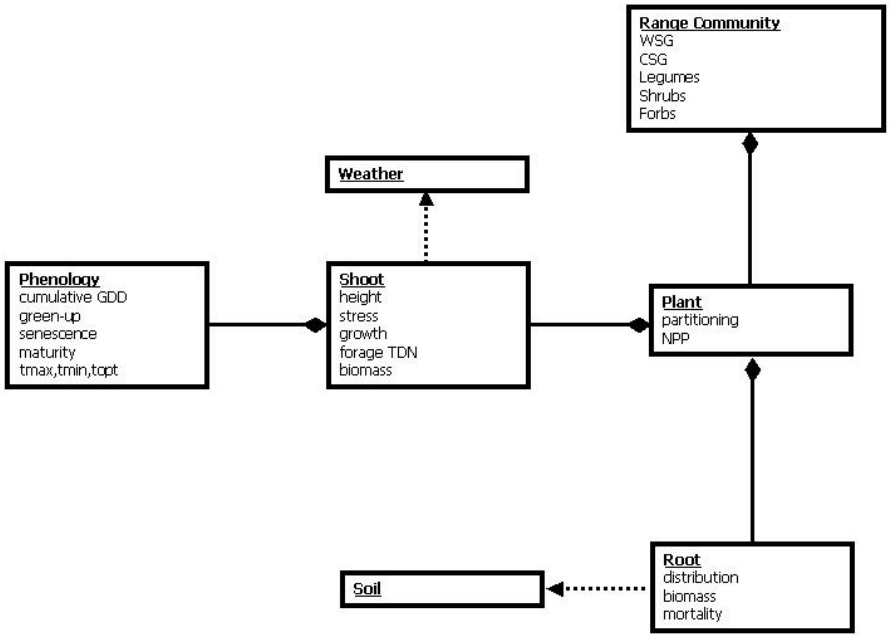

Figure 4. Java classes in the object-oriented forage growth module.

The Java shoot component follows the same basic template as the native OMS forage component shown in the previous section (Fig. 3), except that the execute() method is implemented entirely in Java and does not require the JNI nor a C/C++ bridge. Thus, the Shoot component is only tagged as 'Stateful' but not as 'Native'. The execute() method calls 'runShoot', which subsequently calls Java methods that simulate forage growth processes (e.g. calcStress(), calcdW()). All the methods to simulate processes are contained in the Java class. This is in contrast to an OMS native component, where the process simulations are performed by legacy code and not by the Java class itself.

```
class Shoot implements Stateful {
   public void init() { // Initialization code }

   public void cleanup() { // Cleanup code }

   public void execute() {
     runShoot();
   }
/**
 * Implementation of 'runShoot'
 */
   private void runShoot() {
     calcStress();          //Calculate water and temperature stress factors.
     calcdW();              //Calculate change in biomass (net assimilation).
     calcWg();              //Calculate biomass allocated for growth.
     calcWs();              //Calculate biomass allocated for storage.
     calcSenescent();       //Calculate senescent biomass.
     calcHeight();          //Calculate canopy height.
     calcLAI();             //Calculate leaf area index.
   }
```

Figure 5. Java plant shoot component and its behavior.

Component Attributes

Figure 6 shows a code fragment of the forage component that defines the 'maxGrowthRate' attribute. It is declared as a 'Double', which means that its value is a double precision real number. The attribute has meta data indicating its default value (0.2), its unit (kg/kg), constraints or possible range of values (0.0 to 0.4), and its role as a parameter (constant) in the Forage component. The OMS can compare the current value of 'maxGrowthRate' against the meta data and perform error checking (e.g. value is outside the 'constraint' range) at runtime. The OMS can also test the Forage component independent of other components by executing the code using different values of 'maxGrowthRate' within the specified constraints.

Read and write access of the 'maxGrowthRate' attribute is made possible through the public methods getMaxGrowthRate() and setMaxGrowthRate(), respectively. The setMaxGrowthRate() method requires a specific value of 'maxGrowthRate' as an argument and sets the internal value (this.maxGrowthRate) to it. On the other hand, the getMaxGrowthRate() method returns the current value. Calling these methods is the only way a user can obtain or

change the value of ‘maxGrowthRate’. The ‘set’ and ‘get’ methods for each attribute are properties of a Javabean, which is the component implementation in Java.

```
private Attribute.Double maxGrowthRate;
/**
 * Maximum relative growth rate of the shoot
 * @oms.default 0.2
 * @oms.unit kg/kg
 * @oms.constraint 0.0 .. 0.4
 * @oms.role Parameter
 */
public void setMaxGrowthRate(Attribute.Double maxGrowthRate) {
   this.maxGrowthRate = maxGrowthRate;
}

public Attribute.Double getMaxGrowthRate() {
   return maxGrowthRate;
}
```

Figure 6. Example attribute object (maxGrowthRate) with meta data (prefixed with @oms) and accessors (‘set’ and ‘get’ methods).

OMS Component Builder

The above examples show some details of how components are implemented in the OMS. However, the user does not have to be concerned with component or Java syntax as the OMS provides a graphical user interface (GUI) that includes templates and wizards for building components. The user can simply select the required behavior (stateful, native, etc.), input component meta data, and define component attributes using the GUI while the Java code is generated automatically. The OMS has a FORTRAN 77/95 parser that can be used to generate components from legacy code written in standard FORTRAN 77/95. Once a component has been automatically generated, the user can further edit the component (e.g. add attributes) using the component editor, which is also part of the GUI.

Advantages and Challenges of Component Development in OMS

Given the examples above, it is clear that the developer needs to invest more “up-front” efforts in designing the component structure, I/O requirements, and defining meta data for all the attributes that need to be accessible to the OMS. However, the initial investment in component conceptualization and design can bring numerous benefits once the component is used (and re-used) in different modeling applications. Because OMS components are designed to be context-independent, they can be re-used more readily than code that is monolithic. In addition, the meta data permits introspection of components by the system for runtime checking of parameter values, unit conversion, or automated testing. Meta data also help make the code more readable. The OMS simplifies component creation by providing built-in templates and

wizards, both for native (legacy) and Java code. For a comprehensive discussion of advantages and challenges of developing models using the OMS, the reader is referred to Ahuja et al. (2005).

Conclusion

Future models will be developed by assembling co-operative model components. These units need not necessarily originate from a single modeling group, but conform to a standard interface/protocol for components offering their respective functionality. The assembly of modeling components is aided by the use of model builder tools such as the Object Modeling System (OMS) that extract the self-descriptive information from these components. The Object Modeling System (OMS), as a framework, helps overcome difficulties for future model development by enforcing a standard component design and providing generic software tools to support model development, testing, and deployment. By being able to use existing simulation code written in FORTRAN, the OMS leverages previous investments in legacy science modules and facilitates the development of component-oriented models within a framework that maximizes code reusability and ease of maintenance.

The OMS is free software and is available at http://oms.ars.usda.gov/.

Acknowledgements

We wish to thank Ian Schneider for programming support in the development of the OMS core modules.

References

Ahuja, L. R., J. C. Ascough II, O. David. 2005. Developing natural resource models using the object modeling system: feasibility and challenges. *Advances in Geosciences 3:1-8.*

Andales, A. A., J. Derner, P. N. S. Bartling, L. R. Ahuja, G. H. Dunn, R. H. Hart, and J. D. Hanson. 2005. Evaluation of GPFARM for simulation of forage production and cow-calf weights. *Rangeland Ecology & Management* 58(3):247-255.

Argent, R.M. 2004. An overview of model integration for environmental applications-components, frameworks and semantics. *Environmental Modelling & Software* 19:219-234.

David, O., S. L. Markstrom, K. W. Rojas, L. R. Ahuja, and I. W. Schneider. 2002. The Object Modeling System. In *Agricultural System Models in Field Research and Technology Transfer*, 317-330, Boca Raton, Florida: Lewis Publishers.

David, O., I. W. Schneider, and G. H. Leavesley. 2004. Metadata and modeling frameworks: The Object Modeling System example. In *Trans. 2nd Biennial Meeting of the International Environmental Modelling and Software Society*, 439-443. C. Pahl-Wostl, S. Schmidt, A. E. Rizzoli and A. J. Jakeman, eds.Manno, Switzerland: iEMSs.

Kralisch, S., P. Krause, and O. David. 2004. Using the Object Modeling System for hydrological model development and application. In *Trans. 2nd Biennial Meeting of the International Environmental Modelling and Software Society*, 439-443. C. Pahl-Wostl, S. Schmidt, A. E. Rizzoli and A. J. Jakeman, eds.Manno, Switzerland: iEMSs.