# The Design of a Human Computer Interface for a Multimodeling Object Oriented Simulation Environment

Youngsup Kim and Paul A. Fishwick

Computer and Information Science and Engineering Department
University of Florida

## ABSTRACT

MOOSE (Multimodeling Object Oriented Simulation Environment) is an application framework under the development at University of Florida, which is used for modeling and simulation. MOOSE is based on Object Oriented Physical Modeling (OOPM), and consists of a Human Computer Interface (HCI), Translator, and Engine. A human model author builds the model of a physical system with the help of Graphical User Interface (GUI) and represents his/her model with a picture. The MOOSE GUI contains two types of modelers: conceptual and dynamic. The conceptual modeler supports a model author to define classes and relations among classes in a form of class hierarchy that represents the conceptual model. The dynamic modeler assists a model author to build dynamic models for each of the classes defined in the conceptual model. The dynamic model types supported are Functional Block Model, Finite State Model, Equation Model , System Dynamics Model, and Rule Based Model. We are currently performing research to enlarge the HCI capability by adopting 3D graphics to provide a more immersive and natural environment for better interfacing geometry and dynamic models. We suggest 3D GUI with MOOSE Plug-ins and APIs, a static modeler and 3D scenario. When a user selects a physical object on the 3D graphic window, they use this "handle" to get the conceptual model that relates to that object.

**Keywords:** Simulation, HCI, GUI, Multimodeling, MOOSE

## 1. INTRODUCTION

HCI involves designing computer systems that support people so that they can carry out their activities productively and safely, and its goal is to produce an usable and safe system as well as a functional system. Visibility and Functionality are major roles of HCI.[1]

MOOSE is an acronym for *Multimodeling Object Oriented Simulation Environment*, a modeling and simulation enabling tool under the development at University of Florida. The MOOSE is an implementation of OOPM for multimodeling physical systems into digital models. The MOOSE is composed of **HCI, Translator** and **Engine**. The purpose of the MOOSE HCI is to give model authors a capability of interacting with the MOOSE system to build and execute models with the help of a 2D GUI. Model authors can

- think clearly about, to better understand, or to elucidate a model ,
- participate in a collaborative modeling effort,
- repeatedly and painlessly refine a model as required, in order to achieve adequate fidelity at minimal development cost,
- painlessly build large models out of existing working smaller models,
- start from a conceptual model which is intuitively to domain experts, and to unambiguously and automatically convert this to a simulation program,
- create or change a simulation program without being a programmer, and

Other author information: (Send correspondence to Y. Kim)
Y. K.: Email: yskim@cise.ufl.edu; Telephone: 352-392-1435; Fax: 352-392-1414.
P.A.F: Email: fishwick@cise.ufl.edu; Telephone: 352-392-1414; Fax: 352-392-1414

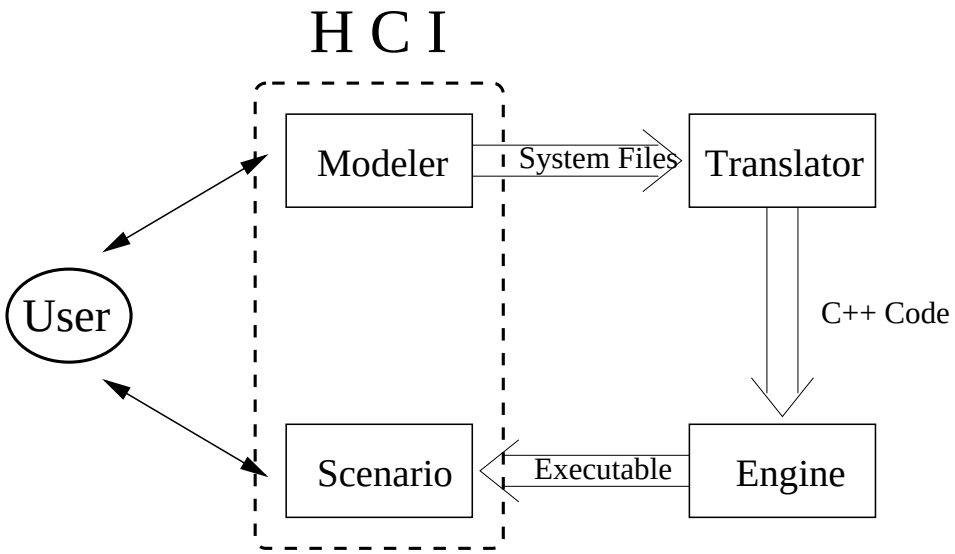



**Figure 1.** The MOOSE components.

- perform a simulation model execution and to present simulation results in a meaningful way so as to facilitate the order objectives above.[2]

The MOOSE HCI consists of **Scenario** and two types of **Modelers**: *Conceptual* and *Dynamic*. A model author can define a model on the modeler, and a model information from the modeler is transfered to the translator. The translator internally converts the information into *Translator Target Language* (TTL) which was written C++. Errors made during a conceptual modeling phase on the Modeler can be detected by the translator. Engine source codes emitted from the translator are compiled and linked directly to create an engine, an executable, by a compiler such as g++ or Visual C++ depending on a platform environment. The scenario displays the results of the model in graphical forms. See Refs. 2,3 for details (Fig. 1). Since a 2D GUI has restrictions for representing and visualizing physical systems, we are evolving the MOOSE HCI capability to 3D graphics to provide a more immersive and natural environment. We will suggest some design issues for integrating 3D geometry and 2D dynamics as a first step toward a 3D MOOSE HCI.

Section 2 describes what kinds of features we are using to support OOPM on MOOSE. Section 3 gives brief introductions about each MOOSE GUI component from the Conceptual modeler to the Scenario, and Section 4 suggests how MOOSE can use 3D graphics for the future.

## 2. OVERVIEW OF OBJECT ORIENTED MULTIMODELING

### 2.1. Object Oriented Physical Modeling (OOPM)

The OOPM is a design approach for structuring multimodels in an object-oriented framework for physical modeling. To build an object oriented digital model for a physical system, we must create a class graph which consists of *classes* and *relations* among the classes. Furthermore, each class has *attributes, methods*, and *objects*.[4] Classes and objects in a digital model being built correspond to those in the physical model being modeled. Objects are instances of a class so that all the objects of one class must keep the same attribute and method structure of the class but can have different instance values according to a purpose of each object.

The relations among the classes are *generalization*, *aggregation*, and *dual*.[4] In the generalization relation, a child class is *a type of* a parent class, and has *Inheritance* property which allows data passing from the parent to the child. In the aggregation relation, a child class is *a part of* a parent class and has *Composition* property which makes data passing from the child to the parent. *Cardinality* of a child class in the aggregation should be specified. In general, without such a specification, it is assumed that a class can be composed of any number of objects of the child class. The dual relation is a combination of the generalization and the aggregation in some specific case. All relations are repeatedly defined in a class hierarchy.

Once a class hierarchy is constructed, we identify attributes and methods for each class. An attribute is either a *variable* or a *static model*. The variable is one of native data types such as integer, real or string, and the static model is a group of objects to represent a geometry of the model. A method can be a code mothod or a dynamic model. The code method is written in a specific object oriented programming language such as C++ or Java. The dynamic method reflects behaviors of the physical model, operates on the static models and variable attributes to effect changes, and employs the multimodeling capability to capture various levels of model abstraction.

### 2.2. Multimodeling

Models that are composed of other models, in a network or a graph, are called a *multimodel*, and a modeling process in which we model a system at multiple levels of abstraction is *multimodeling*.[5] It provides a way of structuring different model types together under one framework so that each type performs its part, and the behavior is preserved as levels are mapped.[6–9] With two adjacent levels in a multimodel hierarchy, components of the higher level can be *refined* by several components of the lower level, and the lower one can be *abstracted* into the higher level. So the higher level takes a role of a black box in which the lower level represents a functional information inside. Of course, the inputs and outputs for both levels should be identical.

## 3. GUI COMPONENTS IN MOOSE

### 3.1. Modeler

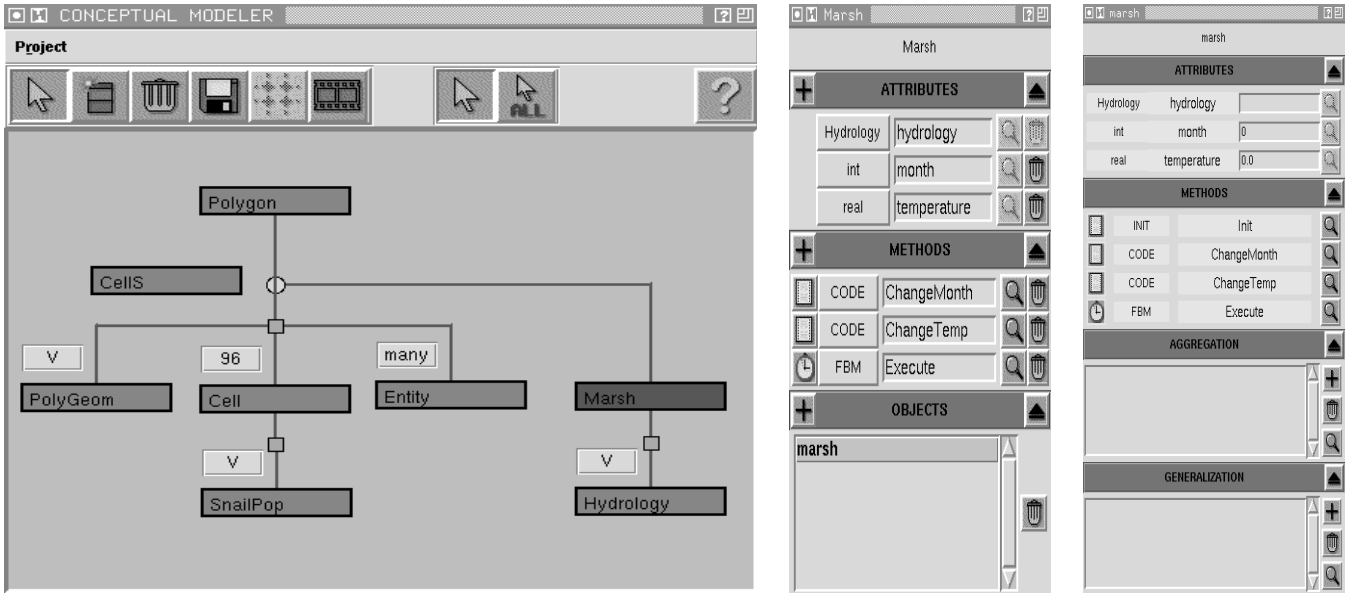
The main purpose of the modeler is for a user to create a digital model of a physical system by defining all the features described in OOPM such as classes and objects. The modeler not only provides users with a set of windows to create and modify models but also produces various topology files for transferring a model information to the translator. The set of windows are divided into two groups: *Modeler window* and *Dialogue window*. A user can draw his/her models on proper modeler windows, and set specific model information on the dialogue windows. The MOOSE windows are written in Tcl/Tk. Once the model construction is completed, the model information is saved into either a class hierarchy definition file, *.hdf*, or a dynamic model topology file, *.tpf* depending on the modeler type. The *.hdf* file keeps information from the conceptual modeler and the *.tpf* files keeps those from the dynamic modelers.

#### 3.1.1. Conceptual Modeler

The major roles of the Conceptual Modeler are to create and manage a conceptual model of a physical system by defining classes, relations among classes, objects, attributes and methods, and to support a control of the whole modeling process from constructing of a model to executing the model.

1. **Windows**
   a user can draw a class hierarchy on a conceptual modeler window. Symbols defined in the conceptual modeler are classes, generalizations, aggregations and the connection links. In Fig. 2, a rectangle represents the class, a small circle represents the generalization, a small square represents the aggregation, a white small rectangle represents the cardinality, and an orthogonal line represents connection links among classes in an aggregation or a generalization relationship. The dialogue windows are *Class window, Object window* and *Method window*. A user can define attributes, methods and objects of a class on the class window of the class. Aggregated classes are automatically set as attributes of the Abstract Data Type on the aggregating class. The object window consists of an attribute section, a method section, an aggregation section and a generalization section. Except for the method section, user can specify specific values on each section. The method window shows input and output parameters of the method and a file name where the method information is located. All the symbols in the modeler window are movable to form a well structured class hierarchy.

2. **Hierarchy definition File (*.hdf*)**
   All the OOPM information and a model execution information should be saved into a fixed *.hdf* file, *project.hdf*. Whenever symbols are created, id_numbers for each symbol type are increased and associated with corresponding symbols, and these id_numbers are used in the *project.hdf* as an identifier. The *project.hdf* includes (1) A model execution information, (2) a class information including attributes (type and name), methods (type, name and *.tpf* file name) and object list, (3) an aggregation/generalization relation information, (4) a toolkit library information (described in the FBM section) and (5) initializations of each object.




(a) Conceptual modeler (b) Class window (c) Object window

**Figure 2.** The Conceptual Modeler in the Snail model.

3. **Control**
   The directory MOOSE includes sub directories: *modeler, trans, engine* and *prj*. Because putting all the models built in MOOSE under the *prj* directory gives more convenient to users as well as MOOSE developers, when the conceptual modeler is opened, a current directory will be automatically changed to a directory of "MOOSE/prj". Of course, user can select any directories on a directory window during model loading and saving.

   In the menu of *project/option*, a user can specify a project name, a project directory, a start object name and a start method name, a simulation time interval and a maximum time limit to control a model execution. Execution control window can be popped up by clicking an image button of a negative film. If a user clicks *delete all* button, the conceptual modeler window is cleared, and an internal data structure is removed so that the conceptual modeler becomes empty.

### 3.1.2. Dynamic Modeler

Methods are divided into dynamic methods and the code methods. The code method is a C++ code segment for the method which is not categorized as adynamic model. It can be written at the conceptual Modeling phase, and saved as a *.cod* file

The dynamic modelers have been developed for each dynamic model type: Functional Block Model (FBM), Finite State Model (FSM), EQuation Model (EQM), System Dynamics Model (SDM) and Rule Based Model (RBM). By the definition of the multimodel, components of a dynamic model can be refined into other dynamic models or code methods. Therefore, all the methods must have multiple input and output parameters so that each outputs of one method are connected with inputs of other methods in the dynamic modeler. Also, one of outputs can be specified as a return value of a dynamic method which cannot have a return statement on it while the code method can explicitly have a return statement. Dynamic modeler windows that are similar to the corresponding windows of the Conceptual Modeler, and produces *.tpf* file keeping topology information of the dynamic method. A name of dynamic method is specified as *Class_name::method_name* (Example: car::Drive means Dynamic method "Drive" of a class "car").

1. **FBM Modeler**

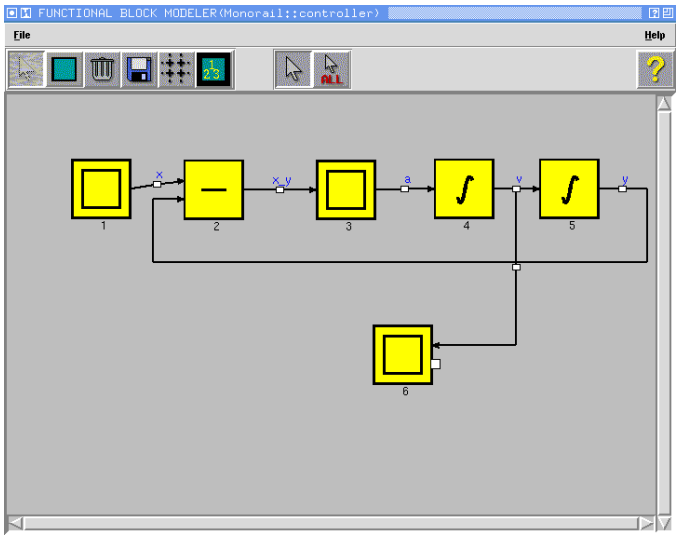



**Figure 3.** The FBM Modeler in the Monorail model.

(a) **Windows**
Symbols of the FBM Modeler are Function Block and Data Flow Link among function blocks. In the FBM Modeler window in Fig. 3, a square represents a function block, and a black line segment between two function blocks denotes the data flow link. When a function block is created, a block number is increased by one and associated with the function block, and a user can specify a method name for the function block. The user can explicitly specify the direction of an input and an output between the blocks, and images can be used for representing complex behaviors of some blocks. The FBM Modeler also supports *Toolkit library window* to select functions in the toolkit library. The toolkit is categorized into groups depending on their properties. *Control* toolkit supports Add, Subtract, Multiply, Divide and Integration, and *Queuing* toolkit helps users build queuing models.

(b) **Event scheduling**
Since an FBM should be executed in a sequential ascending block number order, all the block numbers should be sequential order of execution. Whenever the FBM is saved, an *Automatic renumbering* is performed to renumber block numbers if those are not properly ordered, due to deletions of any blocks or creations of blocks in complicated order. In order to draw functional block models, the FBM Modeler window supports *Grid* facility to place blocks at proper locations and *automatic line adjustment* to redraw all the line segments for a link to be orthogonal line segments by adjusting a start and an end positions.

(c) **Topology File**
The *.tpf* file of the FBM includes function block information, link connections of outputs and inputs among the blocks and external inputs and outputs of the a functional block model. If the toolkit library was used, the toolkit names are detected and saved in a *project.hdf*.

2. **FSM Modeler**

(a) **Windows**
The main symbols of the FSM are State and Transition indicating a movement from one state to another states according to a transition condition. In Fig. 4, a state is represented by a circle, the transition is an arrowed arc, and the condition on the transition arc is a text. When a state is created, a state number is increased by one and associated with the state, and a method name is specified by a user. All the attributes and inputs of the FSM can be used in the condition statement. When saving the FSM, a dialogue window asks for the start state and FSM outputs.

(b) **Event scheduling**
The FSM execution begins at the start state, proceeding to other FSM states according to the result of

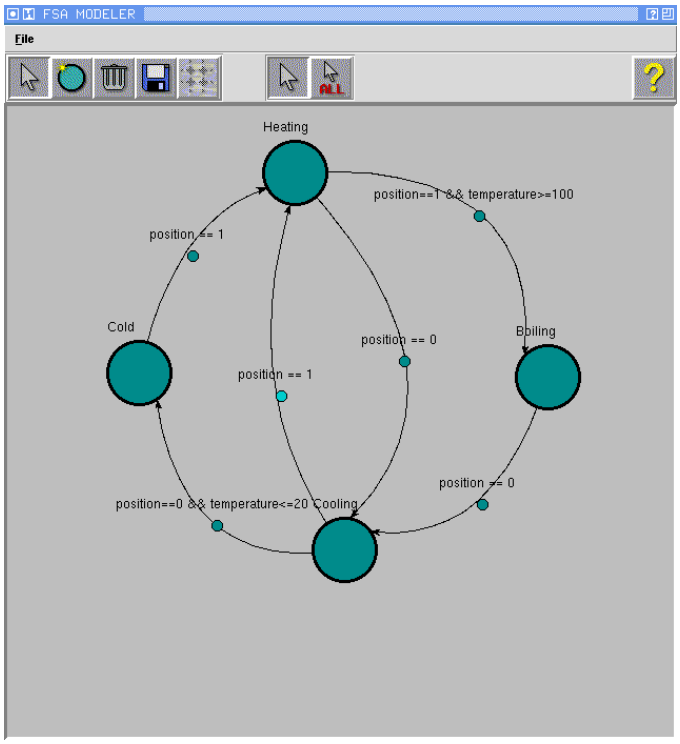



**Figure 4.** The FSM Modeler in the Boiling water model.

transition conditions, and ends when it arrives one of the final states or when the simulation time limit is exceeded.

(c) **Topology File**
The *.tpf* file of FSM includes a state information, transitions and associated conditions among states, outputs of FSM, and the start state of the model. Whenever saving the FSM, state numbers are renumbered to make all the state numbers be in a consecutive order.

3. **EQM Modeler**

(a) **Windows**
The EQM enables the user to create a constraint model using differential and algebraic equations. Any number of $n$th order differential equations may be entered by using an intuitive syntax. Differential equations are represented using symbols such as x, x' as the first derivative of x, x" as the second derivative of x, and x followed by $n$ single quotes as the $n$th derivative of x in general. Several variables can be used, and each variable may be one of attributes in a class to which the equation model belongs and input parameters of the EQM. To give a multimodeling capability to the MOOSE, code methods and dynamic methods are used as a variable. The code methods should explicitly include a "return statement", and dynamic methods must have an output parameter specified as a return value. Then those return values will substitute the method variable in the EQM. If a method has output parameters, equations has to use attributes as actual output parameters of the method. When saving EQM, a dialogue window asks initial values of variables used in the differential equations, and the outputs of EQM. The outputs of the model may be attributes or any order derivative of any variables (Fig. 5).

(b) **Event scheduling**
The EQM executes equations in a top down order. Because every new equation is put at the last line on the EQM window, an order of creating equations is very important. Algebraic equations are executed earlier than differential equations by an action of the translator.

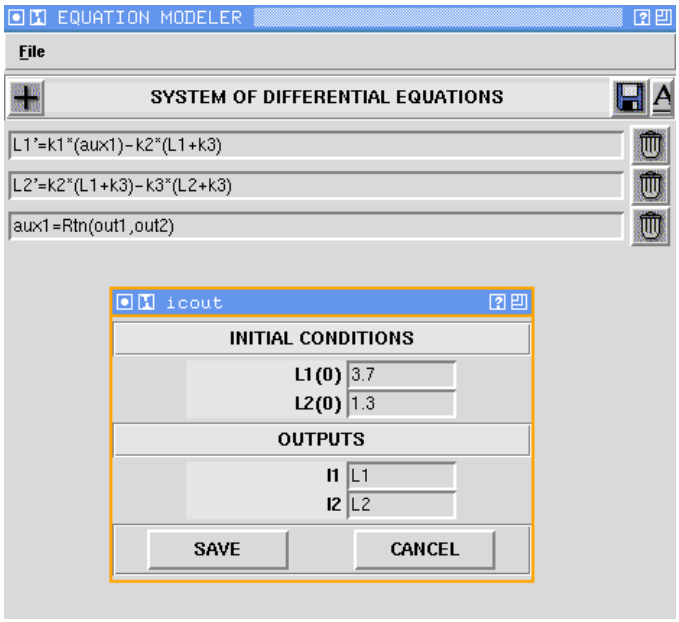



**Figure 5.** The EQM Modeler in the Applesnail model. The dialogue window inside is the initialization window.

(c) **Topology File**
The *.tpf* file of the EQM consists of initial values of derivations of differential equations, *Programmable Form* of equations converted from the equations on the EQM window by *Automatic Equation Conversion program*, and outputs of the EQM.

4. **SDM Modeler**

(a) **Windows**

The SDM is similar to the FBM in the view of a modeler window and an execution behavior. The symbols are Source, Rate, Level, Auxiliary, Constant and Sink. Control flow and Data flow (Cause-and-Effect) are links connecting symbols. In Fig. 6, the control flow is a black arc heading to a center of symbols, and the data flow is a blue arc going to a bottom of symbols. The SDM hass *COMA* ( Class-Object-Methods-Attribute). It displays a list of possible classes from the current class to which the SDM belongs, and shows the corresponding objects, methods and attributes when a user selects one of the available classes. Symbols represent attributes as well as methods. All the inputs are gathered into the source, and the sinks announce the output symbols of the SDM. The SDM supports a dialogue window for specifying a connection of inputs of SDM.

(b) **Topology File**
Our premise is that the semantics of an SDM is equivalent to a set of differential/algebraic equations. Fishwick defines the rules that translate an SDM with a set of equations.[5,10] We convert an SDM into an equivalent EQM[5] by following the rules.

5. **RBM Modeler**

(a) **Windows**

The RBM windows are composed of three sub windows. A *main* window on top shows "IF statements" created and activates other two sub windows to edit the if statements. On the *predicate* window, a user specifies the predicates by selecting variables listed in the *Accessible element list*. A *consequence* window

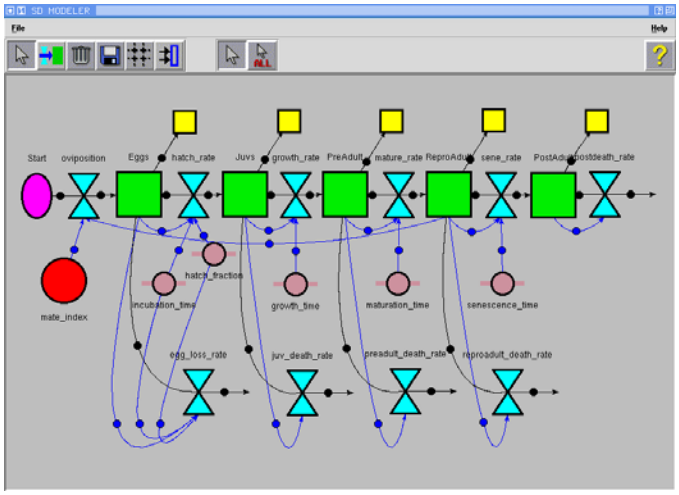



**Figure 6.** The SDM Modeler in the Applsnail model.

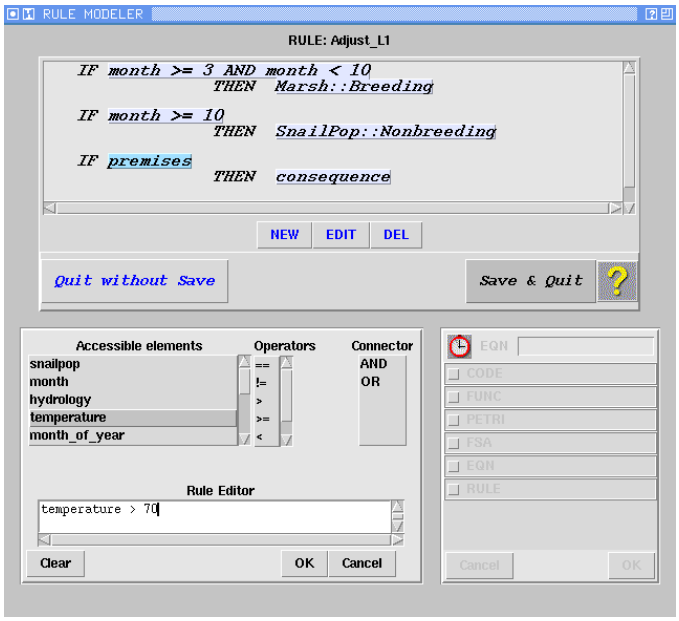
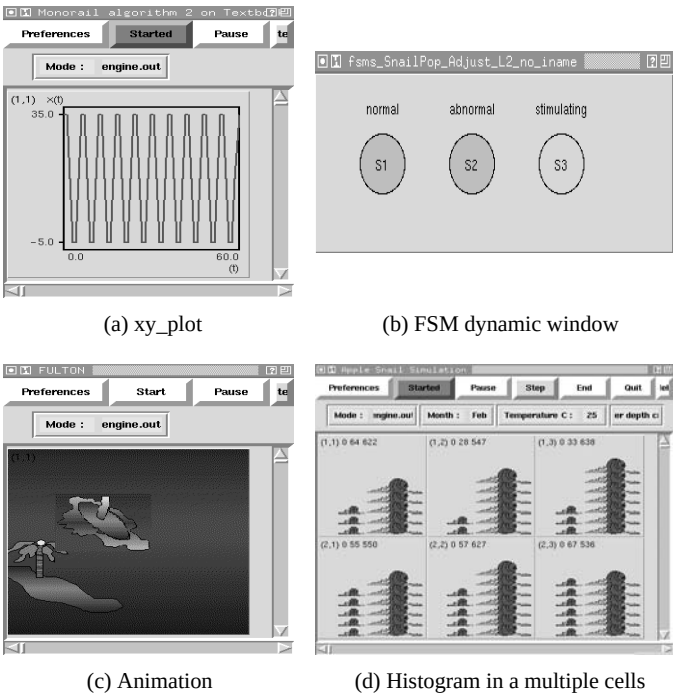



**Figure 7.** The RBM Modeler in the Snail model.




(a) xy_plot (b) FSM dynamic window

(c) Animation (d) Histogram in a multiple cells

**Figure 8.** The Scenario visualization windows.

lists available methods from the class to which the RBM belongs. The *else* part of the IF statement should be converted into a next IF statement. The consequence must be a method and it will support the multimodeling (Fig. 7). RBM evaluates the predicates in a top-down order, and executes a consequence method when a predicate becomes *true*.

(b) **Topology File**
The *.tpf* file of RBM consists of each predicate and consequence in order of creation.

### 3.2. Scenario

While the Modeler is an interface from the users to the MOOSE system, the Scenario is the one from the MOOSE system to the users. It consists of an *Execution control window* and *Output visualization window*. By the Execution control window, a user can translate a model to engine source files, build an engine which is executable code of the model, modify simulation parameters, change a rate of simulation time, and run the engine.

When running the engine, the Scenario establishes a bidirectional pipe connecting it to the engine, and synchronizes an execution of the engine with the scenario. The scenario writes to the pipe as a standard input of the engine, and reads a standard output of the engine from the pipe.[2]

The Output visualization window dynamically shows outputs of the engine in visualization forms: a xy_plot, a Histogram of multiple cells, a 2D Animation on which objects can move around, a FSM dinamics window showing a current states of each FSM Method (Fig. 8). Some simulation outputs are not necessarily amenable to the realtime graphical treatment, and there is a necessary role for traditional methods of analysis.[2,11–13]

## 4. 3D GUI DESIGN ISSUES IN MOOSE

We are enhancing the MOOSE HCI to a more immersive and natural environment by employing a 3D visualization technique. Blending the 3D Geometry model representation with the 2D dynamic model representation requires (1) MOOSE Plug-ins and APIs, (2) Static modeler and (3) 3D scenario.

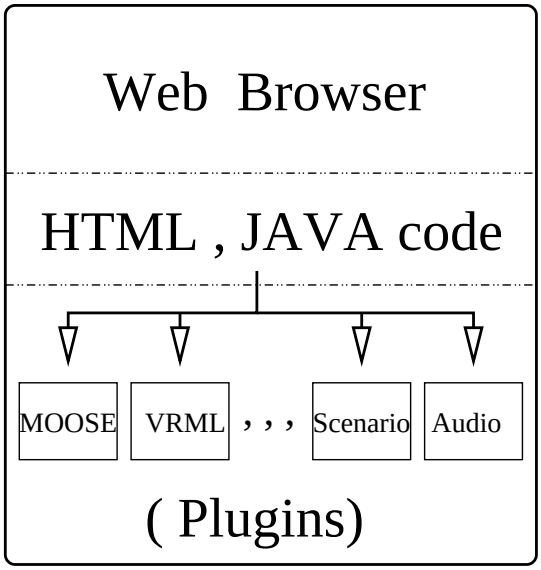



**Figure 9.** The MOOSE environment.

### 4.1. MOOSE Environment

We chose VRML (Virtual Reality Modeling Language, presently VRML2.0)[14] as our 3D graphic programming language. VRML allows to create 3D objects in a 3D space and to add behaviors to the objects. For a communication between the VRML world file (*.wrl*) and its external environment, an interface between the two is needed. The External Authoring Interface (EAI) is the interface defining a set of functions on the VRML brower that the external environment can perform to affect the VRML world. Currently, a Java applet can dynamically communicate with a VRML world embedded in the same HTML page through the EAI. We put the MOOSE on the WEB in a form of a Plug-in with proper APIs to interact with a Java Applets to communicate with VRML world (Fig. 9). In the Web brower, a user can run the MOOSE Plug-in by selecting a MOOSE model embedded in a HTML page, and experience VRML worlds which are dynamically changing according to the results of a model on the MOOSE Plug-in.

### 4.2. 3D Scenario Window

A communication scheme of the 3D scenario and the MOOSE engine is very similar to that of the 2D scenario. As the 2D scenario writes to and reads from the engine through "pipe" built when the 2D scenario activates the engine, the 3D scenario can communicate with the engine via a 3D interface between the two. As discussed in the previous section, VRML worlds can represent 3D scenarios , the EAI serves as the 3D interface between the VRML world and Java, and the Java can communicate with MOOSE by using networking protocols such as TCP/IP. Therefore, the Java is a bridge connecting the 3D scenario and MOOSE.

When a user clicks an object on the 3D scenario window, the EAI passes the object selection information to a Java code, and the Java code sends it to the MOOSE conceptual modeler. A corresponding 2D conceptual model including a class to which the 3D object belongs will be opened so that the user can see a geometry of the object or modify any attributes or methods of the object.

In the other direction, MOOSE can affect the 3D scenario window to reflect outputs of MOOSE execution over time. During a model execution on MOOSE, outputs can be sent to the Java code. The Java code can command the 3D scenario to change its status according to the outputs via the EAI.

### 4.3. 2D Static modeler

The word "static" in the static model refers to the inability of the model to cause changes of values of attributes; it does not mean that the model doesn't change. Our primary type of static model is one that specifies the topology or geometry of a physical object, and the dynamic methods can change the structure of static models over time.[15,16] There are a number of representational techniques for modeling geometry and space, many of which are discussed

by Samet.[17,18,15,16] Our goal is implementing the 2D static modeler to give a 3D static model capability to the MOOSE instead of extending any of static modeling method.

VRML 3D graphics are the results of executions of *.wrl* files. Therefore, the *.wrl* file of the 3D static model can be edited and kept in one of three 2D static modeler types: *VRML code* corresponding to the C++ code method, *Static modeler* corresponding to the dynamic modeler, and *URL* of a *.wrl* file. In any cases, *.tpf* files of static models must keep real VRML codes so that they can be converted into a *.wrl* file, and displayed as 3D static models.

For the VRML code, a user can write VRML programs as the user can write C++ code method in the conceptual modeling. The written VRML codes can be copied directly into *.tpf* file. On the static modeler, to support multimodeling capability of the MOOSE, static models should be aggregated and generalized in a similar way to dynamic models. Therefore, a static model can be refined into another static models. As we map a method name into a function block in a FBM Modeler, each square on a static modeler represents a corresponding static model which can be composed of another static models. The *.tpf* file from the static modeler is a set of VRML codes of all the static models defined in the static modeler. When the static modeler type is an URL, the *.tpf* file can just copy a body of a *.wrl* file pointed to by the URL.

To display 3D static models by using VRML codes, we need enhance the current translator. The translator creates a VRML source file, *.wrl*, by combining each *.tpf* file of static models. Since 3D static models have to communicate with Java code, each 2D static model should be associated with corresponding Java codes including EAI codes. The Java code can be managed in the same ways of the static model: *Java code, Java modeler* and *Java URL*. After translating it, an engine builder must compile and link it to generate Java classes.

## 5. CONCLUSION

The MOOSE HCI supports human model authors in easily building Object Oriented Multimodels of physical systems without being a simulation specialist. Human model authors creates conceptual models and dynamic behaviors of physical systems based on OOPM concepts, create and modify multimodels with different model types, execute them, and finally show the results of model executions. All these activities are assisted by the MOOSE HCI in order for human model authors to represent their thoughts in a form that matched their mental models of their physical domains. When we add 3D graphical capability to the current 2D MOOSE by building the static modeler, 3D Scenario windows and the MOOSE Plug-ins and appropriate APIs on the WEB, the MOOSE will provide more useful functionalities to build a realistic model of complex systems.

## ACKNOWLEDGMENTS

We would like to thank the following funding sources that have contributed towards our study of modeling and implementation of the MOOSE multimodeling simulation environment: GRCI Incorporated (Gregg Liming) and Rome Laboratory (Steve Farr) for web-based simulation and modeling, as well as Rome Laboratory (Al Sisti) for multimodeling and model abstraction. We also thank the Department of the Interior under a contract under the ATLSS Project (Don DeAngelis, University of Miami). Without their help and encouragement, our research would not be possible.

## REFERENCES

1. J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Holland and T. Carey, *HUMAN-COMPUTER INTERACTION*, Addison-Wesley, 1994.
2. R. M. Cubert and P. A. Fishwick, "Moose: An object-oriented multimodeling and simulation application framework," *Submitted to Simulation* , 1997.
3. R. M. Cubert, T. Goktekin and P. A. Fishwick, "Moose: Architecture of an object-oriented multimodeling simulation application framwork," *Submitted to Simulation* , 1997.
4. P. A. Fishwick, "Integrating continuous and discrete models with object oriented physical modeling," *1997 Western Simulation Multi conference* , January 1997.
5. P. A. Fishwick, *Simulation Model Design and Execution : Building Digital Worlds*, prentice Hall, 1997.
6. K. Lee and P. A. Fishwick, "A semi-automated method for dynamic model abstraction," in *Processing of Enabling Technology for Simulation Science, Pare of SPIE AeroSpace '97 Conference*, 1997.

7. B. P. Zeigler, "Toward a formal theory of modeling and simulation: Structure preserving morphism," *Journal of the Association for Computing Machinery* **19(4)**, pp. 742–764, 1972.
8. P. A. Fishwick, "The role of process abstraction in simulation," *IEEE Transaction on Systems, man and Cybernetics* **18**, pp. 18–39, January/February1988.
9. B. P. Zeigler, *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, 1990.
10. Nancy Roberts, Davis Andersen, Ralph Deal, Michael Garet, and William Shaffer, *Introduction to Computer Simulation: A system Dynamics Approach*, Addison-Wesley, Reading, 1983.
11. J. A. Payne, *Introduction to Simulation: programming techniques and methods of analysis*, McGraw-Hill, 1982.
12. A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, 1991.
13. G. S. Fishman, *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley & Sons, 1973.
14. A. L. Ames, D. R. Nadeau and J. L. Moreland, *VRML 2.0 source book*, John Wiley & Sons, New York, N.Y., 1990.
15. P. A. Fishwick, "A visual object-oriented multimodeling design approach for physical modeling," *Submitted to ACM Transaction on Modeling and Computer Simulation* , 1997.
16. P. A. Fishwick, "Extending object oriented design for physical modeling," *ACM Transaction on Modeling and Computer Simulation* , 1996.
17. H. Samet, *Applications of Spatial Data Structures: Computer Graphic, Image processing, and GIS*, Addison-Wesley, 1990.
18. H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1990.