MOOSE: architecture of an object-oriented multimodeling simulation system

Robert Cubert, Tolga Goktekin, and Paul A. Fishwick

Department of Computer & Information Science and Engineering
CSE Room 301
University of Florida
Gainesville, FL 32611-6120

ABSTRACT

MOOSE (Multimodel Object Oriented Simulation Environment) is an enabling environment for modeling and simulation, under construction at University of Florida, based on OOPM (Object Oriented Physical Modeling). OOPM extends object-oriented program design with visualization and a definition of system modeling that reinforces the relation of *model* to *program*. OOPM is a natural mechanism for modeling large-scale systems, and facilitates effective integration of disparate pieces of code into one simulation. Components of MOOSE are Modeler, Translator, Engine, and Scenario: (1) Modeler interacts with model author via GUI to capture model design, (2) Translator is a bridge between model design and model execution, reading Modeler output, building structures representing model, and emitting C++ (or potentially other) code for model; (3) Engine is a C++ program, composed of Translator output plus runtime support, compiled and linked once, then repeatedly activated for Model Execution; (4) Scenario is a visualization-enabling GUI which activates and interacts with Engine, and displays Engine output in a form meaningful to user. Dynamic model types supported include Finite State Machine, Functional Block Model, and Equational Constraint models; alternatively, model authors may create their own C++ "code models"; model types may be freely combined; class libraries facilitate reuse. MOOSE emphasizes multimodeling, which glues together models of the same or different types, produced during model refinement, reflecting various abstraction perspectives, to adjust model fidelity, during development and during model execution. Underlying multimodeling is "Block" as fundamental object. Every model is built from Blocks, expressed in a Modeling Assembly Language.

Keywords: Simulation, Multimodel, Object-Oriented Modeling, Object Oriented Physical Modeling, Visualization

1. INTRODUCTION

MOOSE is an acronym for "Multimodel Object Oriented Simulation Environment", a modeling and simulation enabling tool under development at University of Florida. MOOSE is an implementation of OOPM (Object Oriented Physical Modeling),[1] an approach to modeling and simulation which promises not only to tightly couple a model's human author into the evolving modeling and simulation process through an intuitive HCI (human computer interface), but also to permit a model's author to satisfy any of several objectives: (1) to think about, to better understand, or to elucidate a model; (2) to repeatedly and painlessly refine a model as required, in order to achieve adequate fidelity at minimal development cost; (3) to create or change a simulation program without being a programmer; (4) to start from a conceptual model which is intuitively clear to domain experts, and to unambiguously and automatically convert this to a simulation program; (5) to perform simulation model execution and to present simulation results in a meaningful way so as to facilitate the other objectives above.

Sometimes, modeling alone is enough, such as when the objective is to learn about or better understand a phenomenon or system, or to communicate about such a system with one's colleagues. This aspect of OOPM and of MOOSE should not be overlooked, and would be *per se* justification for the development of MOOSE. But beyond modeling, it is usually the case that a model author wishes to construct a simulation program and perform model execution, for any of various reasons, including : (1) to empirically validate the model based on observed behavior; (2) to select or adjust various parameters and values and observe their effect; (3) to measure performance; (4) to gauge model fidelity and assess its adequacy.

In prevalent practice, a model author makes what is known as a *conceptual model*, often similar to a "blackboard picture" with annotations, and uses this model to describe to one or more programmers detailed requirements for a simulation program to be written, based on the conceptual model. Programmers then write a program, *but there is not necessarily a relation between the conceptual model and the program subsequently produced.* MOOSE offers to improve this situation: MOOSE assists the model author with constructing the conceptual model, and then builds a simulation program in an unambiguous way from the conceptual model. MOOSE thus provides a *mapping* from conceptual model to simulation program. Advantages include: (1) built-in model validation[2]; (2) partial automation of the development process, allowing model authors and programmers to focus on the difficult, rather than on the tedious; (3) easier accommodation to change, leading to a view of change as acceptable instead of as a threat; (4) reducing the response time associated with system development, allowing the model author to effectively drive the development process.

The amount of detail in a model reflects the model author's abstraction perspective.[3] Refinement to greater detail is used to obtain model fidelity that is adequate in the eyes of the model author, from a given abstraction perspective,[4] and with certain objectives for the model or simulation to meet.[5] MOOSE addresses this area with multimodeling, an approach which glues together models of the same or different types, produced during the activity of model refinement, and reflecting various abstraction perspectives.[6] Refinement can be adjustable during model execution as well as during model design. The pieces that are put together to form a model, such as described above, are *dynamic models.* Dynamic model types supported include Finite State Machine (FSM), Functional Block Model (FBM), and Equational Constraint models (EQN); alternatively, users may create their own C++ "code models"; model types may be freely combined. The dynamic model types implemented so far form a popular collection of approaches used in simulation.[7] Additional dynamic model types are certainly in order and will likely be added to the MOOSE repertoire.

Reuse, both by a single model author, and among model authors, is encouraged by availability of class libraries. These we anticipate will form a resource of growing value as MOOSE matures. For example, one model we built, "boiling water" model,[7] is an FSM multimodel, part of which is shown in Figure 3b, and whose Scenario GUI appears in Figure 2b. Later, we implemented a model of Robert Fulton's steamship,[8] whose classes and class relations appear in Figure 1, and whose FBM appears in Figure 3a. When the fulton model was built, the boiling water model's *Pot* re-emerged as a reuse component: it became the *Boiler* of the steamship!

Components of MOOSE are Modeler, Translator, Engine, and Scenario. These components are relatively autonomous, and interact only via flat text files and pipes. *Modeler* interacts with model author via GUI, captures model design, and saves it in a set of files. Modeler eschews dependence on target language or platform. *Translator* is a bridge between model design and model execution: Translator reads the language-neutral model description files produced by Modeler, builds internal structures representing the model, and emits a complete computer program for the model, in Translator Target Language (TTL). Presently MOOSE has one TTL, which is C++; potentially, there may be any number of TTL's (*e.g.*, Java). *Engine* is a simulation program whose source code is written in TTL (*i.e.:* C++). Engine is composed of Translator output plus runtime support. It is compiled and linked once, then repeatedly activated for model execution. *Scenario* is a visualization enabler employing a GUI. Scenario activates and initializes Engine, at the behest of user (who may or may not be model author). Scenario is capable of synchronous interaction with Engine, displaying Engine output in a form meaningful to user, optionally allowing user to interact with model execution.

The balance of this paper is organized as follows : section 2 explains how an Object Oriented approach is used by MOOSE to capture the geometry and dynamics of a model; section 3 explains how MOOSE uses multimodels to facilitate model refinement to achieve appropriate levels of model fidelity; section 4 describes the components of MOOSE in some detail and how they interact; section 5 covers some important MOOSE internal classes; section 6 describes portability and extensibility issues; section 7 is about our conclusions and plans.

2. AN OBJECT-ORIENTED APPROACH TO CAPTURING MODEL GEOMETRY AND DYNAMICS

2.1. An Object Oriented Approach

Taking its cue from OOPM, MOOSE uses an object oriented approach.[9] Classes and objects in the digital world being built correspond to classes and objects in the real world being modeled. This approach has been found to not

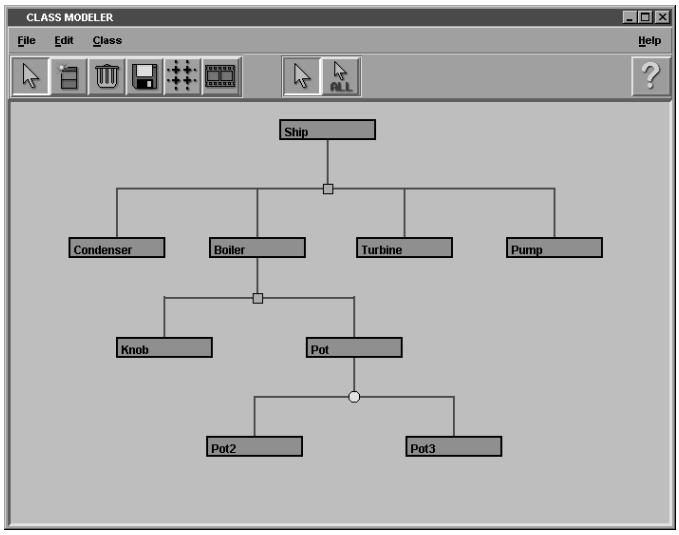


Figure 1. MOOSE Modeler GUI for model of Robert Fulton's steamship, showing classes and relations. Small circles denote specialization; small squares denote aggregation. The Condenser, Boiler, Turbine, and Pump classes in this figure are the same classes as those whose functional blocks appear in Figure 3a.

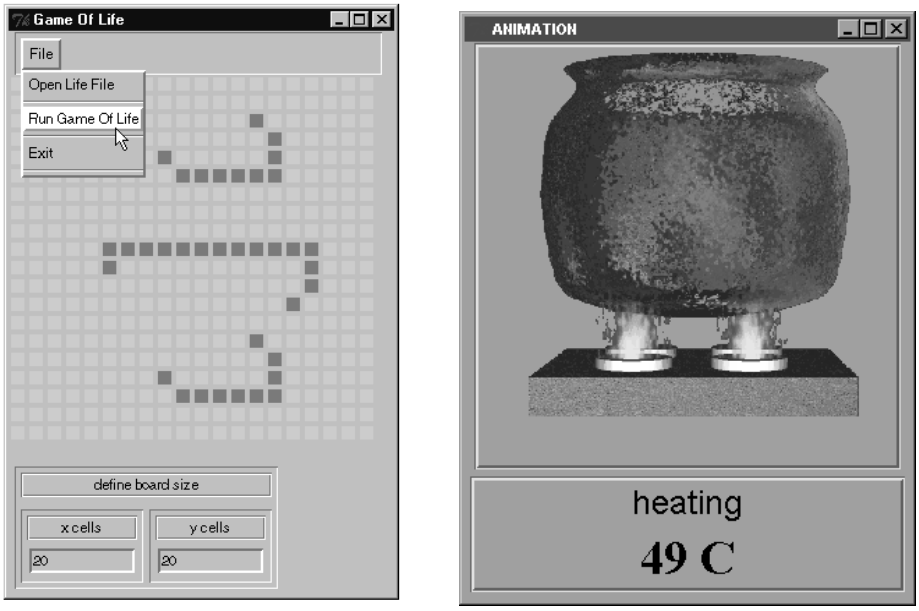
only be intuitively appealing to model authors, but also to be both effective and efficient at capturing the elements of meaning which must be represented in the model.[10]

Not only is object identification performed, but also the model author is encouraged to make explicit the relations among classes. Most notable among these relations are specialization and aggregation. *Specialization* is the "traditional" relationship of derived class (or subclass) to base class, such as "an orange is a *kind of* fruit", whereas *Aggregation* embodies the idea of containment or composition, such as "a car *has* wheels". Specialization has been extensively investigated by many[11]; as has the equivalent concept of *Generalization.*[12] Yet Aggregation or equivalently, *Composition*, examples of which abound in most models we have encountered, and which we have found to be of fundamental importance to the process of modeling, has received less attention; thus, its treatment has received, and continues to receive, keen scrutiny as we develop MOOSE under OOPM principles.

As MOOSE is requiring the model author to communicate relevant object identification and relations, it appears to be (and in fact is) building a conceptual model, which can be a handy representation for the model author. And this kind of "blackboard model" is often useful for one person involved in a project to communicate with his or her co-workers. Yet two other important things are going on: (1) as the model takes shape before his or her eyes, the model author often gains understanding, as represented in the aphorism "the best way to learn something is to have to explain it to someone else"; and, (2) a model description is being constructed which although independent of any programming language, is nonetheless unambiguously and automatically convertible to a simulation program in some programming language, e.g., C++, when the model author wishes to do so. Making explicit the classes and objects, and their relationships often sheds light on what is being modeled and surfaces questions and ambiguities which must be addressed to achieve the modeling or simulation objective. This is part of what is meant by tightly coupling the model author into the modeling and simulation development loop.

2.2. Attributes, Abstract Data Types, Aggregation, and Containers

Attributes are defined for each class. In addition to the "native" data types (integer, real, and string), MOOSE also permits arbitrary abstract data types to be attributes. These abstract data types are just classes like all the other classes in the model, and they play an important role in representing aggregation. We have several ways in MOOSE to represent aggregation. One way is to create the constituent elements as attributes of the aggregate class. A decision as to the best representation rests on answers to questions such as whether the number of items of a constituent type is one, (*e.g*, a car *has* one steering wheel), or more than one (*e.g.* a car *has* four tires); whether the number of items of a constituent type elements is known in advance and fixed (number of cells on a checkerboard),



(a) (b)

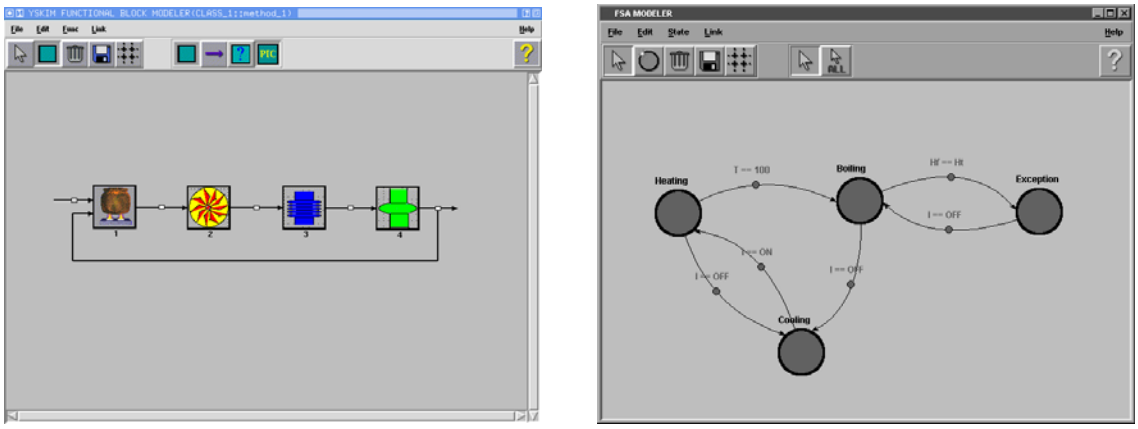
Figure 2. MOOSE Scenario GUI's for two models. On left, Conway's game *Life*. On right, *boiling water* model.

or is inherently variable (a population of deer). When there is only one item of its kind, the class of that item is the type of the attribute. When there are multiple, and especially when there are an uncertain number of items of a kind, an abstract data type which is a *Container class* becomes the attribute. This container class holds elements of the contained type. For example, *Tires*, a tire container, might hold four tires, and this tire container is an attribute of the class *Car*. Alternatively, *DeerPop*, a deer container, might hold any number of deer, and this deer container is an attribute of the class *Everglades*. Container classes have been found to be an effective way to represent one important aspect of aggregation. Provision is made in MOOSE for optional automatic population of containers, and to allow the model author an optional constructor to write custom code to initialize containers.

Another aspect of aggregation is how to relate an attribute of an aggregate class with the corresponding attribute in its constituent classes, when such correspondence exists. An example is biomass in an ecosystem simulation. A deer has a biomass which is its weight. Our deer population in the example above thus has a total biomass which is the sum of the weights of every individual deer. Moving to higher levels of aggregation, the Everglades has a biomass which is the sum of the biomasses of all populations in the model. Here the relation is *summation*. While summation is a popular relation, it is by no means the only relation for aggregation. We are presently studying the phenomenon of aggregation relations in general, and building a repertoire of the most common aggregation relations, such as summation, concatenation, etc. is an active area of work in MOOSE.

2.3. Capturing the Geometry of a Model

MOOSE is based on OOPM, and OOPM in turn has a number of tenets, two of its most important relate to geometry and dynamics. Geometry relates to space, and Dynamics has to do with temporal evolutions. We first discuss MOOSE model geometry. When a simulation involves a world where entities interact and evolve over a field, with the field often influenced and changed by the presence and activities of the entities, one usually thinks of model geometry in the conventional sense of defining properties of the space over which the field is defined and through which the entities move. This is certainly one form of model geometry, and one which MOOSE supports. An example of this kind of simulation is John H. Conway's board game "Life" ,[13,14] which has been implemented



(a) (b)

Figure 3. Two of the MOOSE Modeler GUI dynamic model editors. On left, FBM editor, showing part of *fulton* model, with functional blocks for Boiler, Turbine, Condenser, and Pump. On right, FSM editor, showing part of *boiling water* model, with states for Heating, Cooling, Boiling, and Exception; predicates defining FSM state transitions appear as text near the tiny circles on arcs. Boiling water model is one component of fulton model; thus, FSM in (b) is actually inside the Boiler functional block of FBM in (a).

as a MOOSE model. A complete explanation of the game is not feasible here, but the interested reader can learn details from the references .[15] Summarizing the model, the *Game* is an aggregation of, or "has a", *Board* and *Rules*. Board in turn has a container *Cells* and a *BoardGeometry*. Cells container in turn contains many individual *Cell* objects. BoardGeometry maps the real location or identity of Cell objects in the Cells container onto 2-dimensional space. Rules tells the Game how to take each cell on the board from one tick of the simulation clock to the next (*e.g.*, birth, death). This illustrates that MOOSE can model *any* space by mapping elements of a container class of individual region objects onto a space of any specifiable complexity. MOOSE considers geometry not only in the narrow conventional interpretation above; but also, in a broader sense, the space under consideration can be a space other than a physical space; rather, it can be a space over which classes and/or objects relate. MOOSE is capable of modeling this sort of geometry as well, through class definitions and relations among classes.

2.4. Capturing Dynamic Behavior of a Model

Classes would be uninteresting indeed without methods. In MOOSE, these detailed aspects of every class may be readily added, changed, and removed, as part of model development, at any time. Dynamic behavior of the system is represented in class methods. Here is where MOOSE makes good on its promise to the model author to be able to create or change a simulation program without being a programmer. MOOSE presently incorporates several kinds of "dynamic model": FBM, FSM, and EQN, with others contemplated, such as Petri nets, and Rule based dynamic models. From this ensemble of popular and capable dynamic model types, the model author picks one or more dynamic model types to define methods of the classes of the model. Construction of each specific dynamic model typically involves drawing the kinds of "pictures" that people tend to make on the back of an envelope or a blackboard when informally describing a model to someone else. The MOOSE HCI facilitates these constructions: allowing the model author to specify components, connect components, provide inputs, outputs, conditions, and so forth.

Although promising models without programming, MOOSE also tries to be tolerant and flexible. Thus, if none of the dynamic model types suits, the model author is free to write what are termed "code models" or "code methods". A code method is a function body written in the TTL used for the MOOSE system. Presently this language is C++. MOOSE design ideology suggests that code methods be the exception rather than the rule. Typical use of a

code method is to provide that one small piece of some models that cannot be described using the available dynamic model types, and to rely on dynamic models for the rest, in a way that is analogous to construction of an Operating System kernel in a high level language, with just a few assembly-language routines where needed.

3. FACILITATING MODEL REFINEMENT

Constructing a model is almost always an iterative process, with model structure taking on a tree-like appearance. The broadest description of the model is like the root of a tree. One then typically decomposes this broad but nebulous description into subordinate parts, each part being a refinement of the model in the broad description statement. The result typically is a tree with some leaves near the root, and others farther "down". Thus the level in the tree is related to the level of abstraction which one associates with thinking about and describing the model.

Resources are limited, and by this we mean both model development resources and simulation runtime resources. Thus one typically refines using a breadth-first approach, and this tree-like structure accordingly takes on an uneven shape, with some parts of the tree being of greater height, and others being of shorter height, reflecting the underlying decision criterion, which is to refine only as much as needed to achieve required levels of model fidelity. But knowing what is needed to achieve required levels of model fidelity often requires iterating through several model designs, and even measuring performance of the model execution. Multimodeling can be used in the development process, to conserve valuable development resources, including time, by limiting the depth of some subtrees; and to provide a top-down skeleton within which development may proceed. A rude shallow model can be run, and analysis can pinpoint those model subtrees where additional fidelity is needed. This is an adaptive mechanism to focus and guide development. The evolving model is thus its own prototype. It needn't be discarded, as in throw-away prototyping, nor does it suffer the chaos that often accompanies the "exploratory prototyping" or "exploratory programming" approach.[2]

Thus MOOSE provides facilities for multimodeling, a term coined by Oren[16] later elaborated by others,[17,18] and by which we mean model refinement into more detailed component models, reflecting a number of abstraction perspectives.[6] This is a very intuitive concept: most people multimodel without thinking about it; yet, they *do* benefit from encouragement in this direction, and especially from automatic management of the resulting complexity. The multimodel definition is recursive: refinement proceeds as far as needed. By facilitating this kind of work and encouraging this kind of thinking, MOOSE contributes to management of the model: extending (or reducing) model refinement at any stage of the game. Typically, refinement is extended when fidelity is inadequate, and is reduced when the simulation takes too long to execute. Now having explained multimodeling in general, we proceed to a specific taxonomy. There are two perspectives from which one may look at multimodeling: time of binding and dynamic model type. Each perspective leads to a dichotomy. The overall result is a small taxonomy which will now be presented.

Multimodel dichotomy based on time of Binding: In a temporal sense, regarding the time at which the level of refinement is bound or fixed, MOOSE recognizes two kinds of multimodel: static and dynamic. *Static Multimodels*: The evolution of model refinement that takes place over the model development life cycle results in a static multimodel; that is, we change the model's structure from time to time, but whenever we build a simulation program representing the model, we freeze model structure as of that time. Of course we can change it later and build a new simulation program, but the static multimodel persists until this is done. *Dynamic Multimodels*: The second kind of multimodel is the dynamic multimodel, and the MOOSE runtime environment supports this kind of multimodel too. A dynamic multimodel changes its refinement on the fly, in response to system constraints. A typical constraint is a realtime constraint on when the simulation must complete. Presently, MOOSE does not provide the executive logic which decides when to change refinement depth; but, given such logic, MOOSE has the capability to reconfigure model refinement on the fly. Others are presently working on providing this logic for MOOSE.[8]

Multimodel dichotomy based on type of Dynamic Models: When a model is refined, each level is usually described by one or more dynamic models. Each dynamic model is of some type, e.g., FSM, FBM, etc. If all the dynamic models are of the same type, then the multimodel is *homogeneous*. If the dynamic models are of different types, then the multimodel is *heterogeneous*.

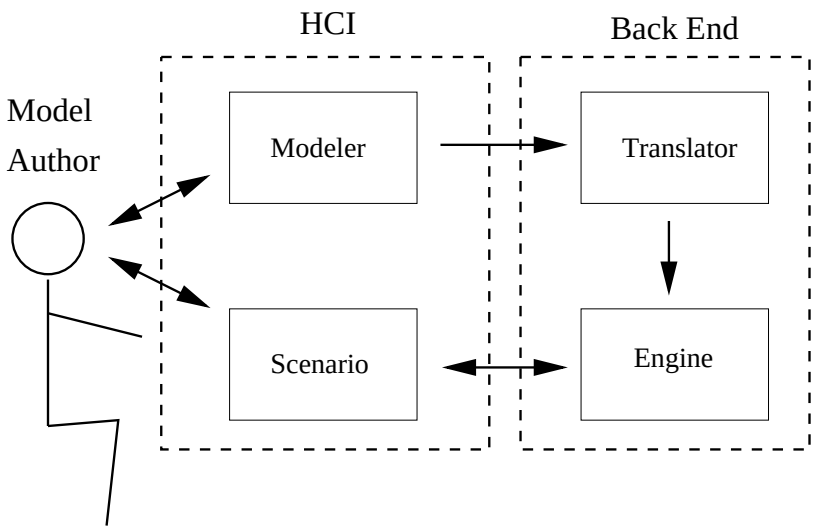


Figure 4. The parts of MOOSE: Modeler and Scenario form the HCI. Translator and Engine form the back end. Information flows in the directions indicated by the arrows.

4. THE PARTS OF MOOSE

MOOSE is comprised of four components: Modeler, Translator, Engine, and Scenario. Modeler and Scenario form the "front end" or HCI, and Translator and Engine form the "back end". The parts are relatively autonomous, and for a number of reasons, communications between the parts of MOOSE is by way of text files and pipes.

4.1. Modeler

Modeler has the GUI (graphical user interface) which interacts with the model author to construct a model. Modeler's GUI has several parts: (1) one for defining classes, objects, relations among classes, the attributes and methods of each class, and the parameters of each method; (2) another for each dynamic model type, thus, there is an FSM editor, an FBM editor, and an EQN editor. Intermediate results can be saved and later reloaded from disk files. Output in the form of a language-neutral model description is sent to a number of flat text files for use by Translator.

4.2. Translator

Translator receives input in the form of a model description, which is a set of flat text files. The model description is ordinarily one that was produced by Modeler; alternatively, a substantial amount of testing and model development was done by creating model descriptions "by hand" with a text editor. Translator works in two phases:

Phase 1: scan, parse, and build : Phase 1 consists of scanning and parsing the model description, and constructing an internal representation of the model, including: classes, instances, relations among classes, name and type of each attribute and each method of each class, initial value of attributes, names and types of parameters of each method, and where to find details about each dynamic model specified as a method.

Phase 2: emit code in Translator target language : Phase 2 involves emitting the code for the simulation program representing the model, in a given target language. Presently the Translator target language (TTL) is C++; however, Translator is designed so that its code-emitting component can in future be replaced with a component to support some other Translator target language, such as Java. Because phase 1 forms about 2/3 of Translator and phase 2 forms only about 1/3, this two-phase approach is an example of the reuse principle of the object-oriented paradigm applied in MOOSE.

Translator's output, as previously mentioned, is a complete "Engine" program in written in TTL which is C++ (including indentation and comments); specifically, (1) a header file, consisting primarily of class declarations, and (2) a source file, containing C++ translation of each dynamic model method and each code method, as well as code to invoke engine runtime support.

4.3. Engine

Translator's output is an Engine source program. The next step is to translate the Engine to create an executable. In MOOSE this is done with a "make" utility program; alternatively, Engine can be made directly by a compiler such as Visual C++ or g++. At link time, a number of runtime support components are added from object libraries, the most important of which is *ooSim*.[19]

The ooSim event scheduling toolkit: All dynamic models are translated into C++ code which relies on the underlying event-scheduling of the *ooSim* dispatcher for propagating event chains. *ooSim* is an event-scheduling simulation queuing model toolkit which arose as an object oriented re-implementation and extension of the SimPack toolkit[7]; SimPack is, in turn, based on SMPL.[20] In addition to event scheduling, ooSim also provides numerous other forms of support, such as pseudo-random number generation, etc. But it is the event scheduling that is ooSim's primary support role.

Engine source file contains code to initiate one or more event chains. These event chains propagate independently of one another, and the time step of each chain is independent of the time step of every other event chain. The event scheduler propagates each event chain until that event chain terminates itself, or until the simulation clock reaches the overall time limit specified for the simulation in the model definition.

4.4. Scenario

Once Engine is built, it can be run as many times as desired. This done under auspices of Scenario. Scenario establishes a bidirectional pipe connecting it to Engine. The effect of this connection is to activate Engine, and then to synchronize Engine's execution with that of Scenario. This is an interactive connection, which can be used to adjust the rate of progress of Engine. Engine can be allowed to free-run, or it can be made to single step, or to run at any pace in between. This is useful for generating animations with which the model author can interact. Time scales scan be stretched or compressed. Things which ordinarily happen blindingly fast can be slowed down. The rate of progress can be adjusted to focus on parts of the simulation execution that are of particular interest.

The bane of simulation, for which it receives bad press, is output in the form of reams of computer printout. This situation is improved by using Scenario. Scenario has a GUI of its own, with which model author interacts. Scenario can initialize parameters and pass them to Engine. Most important, Scenario filters Engine's output. Scenario takes that dull boring simulation output from Engine, and turns it into appealing, meaningful, usually graphical, output. Engine is thus left free to do what it does best: model execution, producing those important although (alas) dull answers. Scenario then does what it does best: facilitating visualization.

Although Scenario detail is unique to each model, we are working on a toolkit of visualization instrumentation for reuse. This kit includes dials and gauges, like those seen in an automobile or those which measure progress when installing software, as well as simple xy plot graphs. Yet, some simulation output isn't necessarily amenable to graphical realtime treatment, and there is a very necessary role for traditional methods of analysis.[21–23] MOOSE can support this in two ways: Engine can send output for this purpose to a file separate from that examined by Scenario; alternatively, Engine can route all its output through Scenario, and Scenario can direct some of that output to the user, and other output to a file. Further analysis can then be handled by software accessible through Scenario, or completely external to MOOSE.

5. KEY CLASSES IN MOOSE BACK-END SOFTWARE

Blocks: This section describes some key classes in MOOSE back-end software (back end is comprised of Translator, TTL, and Engine). First we discuss the *Block* class: MOOSE models are multimodels built mostly of dynamic models, and every dynamic model has a structure (subordinate elements) and a topology (how those subordinate parts are connected). FSM's, for example, have states connected by arcs labeled with predicates that control state transitions; and, FBM's, for example, have function blocks connected by traces which carry output of one block to input of another. In MOOSE, every subordinate element of every dynamic model (*e.g.*, state of an FSM, functional block of an FBM, etc, is an object of a derived class of the base class *Block*, so called for historical reasons (our first dynamic model type was FBM). This homogeneity facilitates model refinement and so is an underlying support for multimodels of all kinds, most especially heterogeneous multimodels.

Clusters: Associated with each Block object is a structure known as Clusters, which hold pointers to objects known to belong to classes that have certain relations to the object. Each dynamic model method belongs to a class, but the identity and true nature of each block within that dynamic model can be bound as late as every time the method is dispatched. Clusters are searched as needed to identify the block objects to associate with each element of a dynamic model, just before each execution of the dynamic model. This dynamic block binding facilitates dynamic multimodeling. It is also anticipated to support distributed simulation when MOOSE moves onto the web.

Context: MOOSE engine is event-scheduled using ooSim. This is essentially transparent to the model author, who only specifies the time step for each model within each event chain (or one overall time step if all time steps are the same). Event chains can terminate themselves, or they can end when the simulation clock reaches a specified time. The consequence of event scheduling is that all events, such as one execution of the code of a dynamic model are called from the ooSim event Dispatcher. There cannot be any loops in the dynamic models: the equivalent of loop behavior is attained by propagating event chains. Each dynamic model is a method of some class. ooSim does not use global symbols, so to have the equivalent of static local variables *private to each object*, a *Context* structure holds this information. Contexts are generated automatically by Translator for dynamic models. This facilitates event-scheduling.

Glist: MOOSE needed a base class for lists, because MOOSE has lots of lists to manage. The *Glist* class is this base class. A Glist object is a dynamically allocated array of pointers. When insertion is performed, the array senses when it is full, and automatically expands, in a way transparent to the caller. Glist is not a linked list, it is an array, so it has speed and safety advantages relative to linked lists. Derived classes of Glist are made type-safe [24] by appropriate casts in derived class declarations (*not* in calling code!). There are specialized methods that were needed for one derived class or another, and were put into the Glist base class, and so became available for all derived classes, present and future, to use. First Glist was handy in Translator, then it was reused in Engine runtime support; most recently, it has become the foundation of the container classes which facilitate representing aggregation.

Dynamic: The present MOOSE implementation includes several of dynamic models: FSM, FBM, and EQN. We see needs for other kinds of dynamic models, such as Petri nets, Rule based models, Fuzzy models, and perhaps others. There is a requirement that MOOSE be painlessly extensible to new dynamic model types. The *Dynamic* class is an abstract base class[11] in Translator, from which all current dynamic model classes internal to Translator are derived, and which will facilitate creation of new dynamic model types in future.

6. PORTABILITY AND EXTENSIBILITY

Portability: Platforms and Languages: We chose two target platforms for MOOSE: the first is Sun's Solaris flavor of System V Unix; the second is MS Windows NT. The Unix platform was chosen for convenience, as it is ubiquitous in our Departmental environment, as well as across academia. The Windows NT platform was chosen because it runs not only on the IBM-compatible PC with Intel x86 CPU, but also on RISC processors like Digital's Alpha AXP, the PowerPC, and MIPS RISC.[25] For MOOSE programming languages, we chose TclTk for our components with GUI's (Modeler and Scenario); and, C++ for our back end components (Translator, TTL, and Engine runtime support). MOOSE back end code has been compiled under MS Visual C++, Borland C++, Borland Turbo C++, and g++.

Extensibility to Distributed Operation: MOOSE is headed for the web. There are two kinds of distributed operation to consider: one is where class definitions are distributed, with one class defined here, another there; the second is where model execution proceeds as a distributed simulation, executing simultaneously on a number of hosts, with one object instantiated here, another there. The MOOSE architecture, and its present implementation, are extensible in this regard. In anticipation of the first of these, permitting distributed definitions of the classes and objects, MOOSE Modeler supports a URL (Universal Resource Locator) type. In anticipation of the requirement to support distributed model execution, the existing framework in Translator and Engine of virtual functions,[11] the *BLOCK* language, Contexts, and Clusters (discussed below), not only support multimodels, but also are intended to

provide this extensibility to support distributed model execution. Just as block definitions can be replaced on the fly in dynamic multimodeling to vary model refinement, so also block definitions can be replaced on the fly to include remote procedure calls (RPC) or equivalent.

7. CONCLUSIONS AND PLANS

TclTk is our programming language for Modeler and Scenario GUI's. As TclTk neither enforces nor facilitates object-oriented methodology, we are looking at ways to retain the benefits of TclTk while improving the reusability and extensibility of the code. We will also explore object-oriented alternatives to TclTk. Scenario has a difficult job: it must facilitate visualization even though every model is different in surface appearance. Scenario is also presently written in TclTk so the remarks above regarding lack of object-oriented support apply as well. Nonetheless, we are working on building a toolkit of popular and reusable dials, gauges, graphs, and clip art, and are thinking hard about ways to generalize Scenario. All our HCI work needs constant review and improvement, as new (especially immersive) technologies beckon. It is a fundamental tenet of MOOSE and OOPM, that the HCI must fit the model author like a glove. Our objective is to make it *fun* to use the MOOSE HCI.

The present MOOSE implementation includes several kinds of dynamic models: FSM, FBM, and EQN. We see needs for other kinds of dynamic models, such as Petri nets, Rule based models, Fuzzy models, and perhaps others. Aggregation and the implications of aggregation pose difficult questions. We have been working on these, and have reached some significant results. More work lies ahead. Solutions in this area will likely have powerful implications. We also plan to take MOOSE onto the web, to distribute objects and perhaps model execution. We plan to evaluate Java and other languages in this context.

ACKNOWLEDGEMENTS

We would like to thank the following funding sources that have contributed towards our study of modeling and implementation of a multimodeling simulation environment for analysis and planning: (1) Rome Laboratory, Griffiss Air Force Base, New York under contract F30602-95-C-0267 and grant F30602-95-1-0031; (2) Department of the Interior under grant 14-45-0009-1544-154 and the (3) National Science Foundation Engineering Research Center (ERC) in Particle Science and Technology at the University of Florida (with Industrial Partners of the ERC) under grant EEC-94-02989. We also acknowledge with thanks the help of Kangsun Lee in providing assisting with LaTeX issues, and of Youngsup Kim and Doug Dillard with figures of the FBM and Life Scenario, respectively.

REFERENCES

1. P. A. Fishwick, "Extending object oriented design for physical modeling," *ACM Transactions on Modeling and Computer Simulation* , July 1996. Submitted for review.
2. I. Sommerville, *Software Engineering, Fourth Ed.*, Addison-Wesley, 1992.
3. P. A. Fishwick, "The role of process abstraction in simulation," *IEEE Transactions on Systems, Man and Cybernetics* **18**, pp. 18 – 39, January/February 1988.
4. P. A. Fishwick, "Abstraction level traversal in hierarchical modeling," in *Modelling and Simulation Methodology: Knowledge Systems Paradigms*, B. P. Zeigler, M. Elzas, and T. Oren, eds., pp. 393 – 429, Elsevier North Holland, 1989.
5. V. Berzins, M. Gray, and D. Naumann, "Abstraction-based software development," *Communications of the ACM* **29**(5), pp. 402 – 415, 1986.
6. P. A. Fishwick and K. Lee, "Two methods for exploiting abstraction in systems," *AI, Simulation and Planning in High Autonomous Systems* , pp. 257–264, 1996.
7. P. A. Fishwick, *Simulation Model Design and Execution : Building Digital Worlds*, Prentice Hall, 1995.
8. K. Lee and P. A. Fishwick, "Semi-automated method for dynamic model abstraction," in *SPIE Conference Proceedings*, 1997.
9. G. Booch, *Object Oriented Design with Applications*, Benjamin Cummings, 1994.
10. B. P. Zeigler, *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, 1990.
11. B. Stroustrup, *The C++ Programming Language, second edition*, Addison-Wesley, 1991.

12. G. Booch and J. Rumbaugh, *Unified Method for Object-Oriented Development*, Rational Software, 1995.
13. M. Gardner, "Mathematical games," *Scientific American, Oct 1970* , 1970.
14. M. Gardner, "Mathematical games," *Scientific American, Feb 1971* , 1971.
15. M. Gardner, *Wheels, Life, and Other Mathematical Games*, publisher, 1983.
16. T. I. Oren, "Dynamic Templates and Semantic Rules for Simulation Advisors and Certifiers," in *Knowledge Based Simulation: Methodology and Application*, P. Fishwick and R. Modjeski, eds., pp. 53 – 76, Springer Verlag, 1991.
17. P. A. Fishwick and B. P. Zeigler, "A Multimodel Methodology for Qualitative Model Engineering," *ACM Transactions on Modeling and Computer Simulation* **2**(1), pp. 52–81, 1992.
18. P. A. Fishwick, "A Simulation Environment for Multimodeling," *Discrete Event Dynamic Systems: Theory and Applications* **3**, pp. 151–171, 1993.
19. R. M. Cubert, "The oosim object oriented simulation library," Tech. Rep. TR951230, University of Florida CISE Simulation Group, 1995.
20. M. H. MacDougall, *Simulating Computer Systems : Techniques and Tools*, MIT Press, 1992.
21. J. A. Payne, *Introduction to Simulation : programming techniques and methods of analysis*, McGraw-Hill, 1982.
22. A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, 1991.
23. G. S. Fishman, *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley & Sons, 1973.
24. B. International, *The World of C++*, Borland International, Scotts Valley, CA, 1991.
25. K. Siyan, *Windows NT Server*, New Riders, 1995.

AUTHORS' BIOGRAPHIES

Robert Cubert is a PhD student in CISE at University of Florida. His research interest is object-oriented distributed simulation. He holds BS degrees in EE from MIT and in Zoology from University of Oklahoma, and an MS in Computer Science from University of Oklahoma. He spent 3 years on Computer Science faculty at California State University, Sacramento, and has a decade of industry experience writing software for realtime control systems and proprietary data communications.

Tolga Goktekin is an M.S student in the Department of Computer and Information Sciences and Engineering at the University of Florida. He received the B.S degree in Computer Engineering from Bogazici University, Turkey in 1994. His research interests are simulation, visualization and user interface design. Tolga Goktekin's WWW home page is http://www.cise.ufl.edu/ tgokteki and his E-mail address is tgokteki@cise.ufl.edu.

Paul A. Fishwick is an associate professor in the Department of Computer and Information Sciences at the University of Florida. He received the BS in Mathematics from the Pennsylvania State University, MS in Applied Science from the College of William and Mary, and PhD in Computer and Information Science from the University of Pennsylvania in 1986. He also has six years of industrial/government production and research experience working at Newport News Shipbuilding and Dry Dock Co. (doing CAD/CAM parts definition research) and at NASA Langley Research Center (studying engineering data base models for structural engineering). His research interests are in computer simulation modeling and analysis methods for complex systems. He is a senior member of the IEEE and the Society for Computer Simulation. He is also a member of the IEEE Society for Systems, Man and Cybernetics, ACM and AAAI. Dr. Fishwick founded the comp.simulation Internet news group (Simulation Digest) in 1987, which now serves over 15,000 subscribers. He was chairman of the IEEE Computer Society technical committee on simulation (TCSIM) for two years (1988-1990) and he is on the editorial boards of several journals including the ACM Transactions on Modeling and Computer Simulation, IEEE Transactions on Systems, Man and Cybernetics, The Transactions of the Society for Computer Simulation, International Journal of Computer Simulation, and the Journal of Systems Engineering.