Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

A FRAMEWORK FOR DISTRIBUTED OBJECT-ORIENTED MULTIMODELING AND SIMULATION

Robert M. Cubert
Paul A. Fishwick

Department of Computer and Information Science and Engineering
University of Florida
CSE Building, Room E301
Gainesville FL 32611-6120, U.S.A.

ABSTRACT

We have developed a multimodeling object-oriented (OO) simulation environment (MOOSE), which is a framework for modeling and developing simulation software. Its architecture derives from Object Oriented Physical Modeling (OOPM), which extends classical object-oriented methodology to allow attributes and methods to take on *models* as values. The MOOSE Model Repository (MMR) allows distributed model definitions, and so supports "web-based simulation", integrated with the web and made available on the Internet. MOOSE features multimodeling, an OO approach to model refinement and abstraction, allowing creation of heterogeneous hierarchical models. Dynamic models comprising multimodels include Finite State Machines, Functional Block Models, Equation Constraint Models, and Rule Based Models. MOOSE emphasizes visualization, & effective use of OO metaphors to connect conceptual model to program, and to capture model geometry and dynamics. The MOOSE human-computer interface has two GUI's: *Modeler*, for model design, and *Scenario*, for model execution control and visualization. MOOSE back end generates a model description in a target language such as C++, then translates and adds runtime support to form an Engine. Model execution consists of Engine running synchronously with Scenario. The MOOSE approach facilitates model development, models with greater intuitive appeal, communication among model authors, better agreement between simulation programs and their conceptual models, component reuse, and model/program extensibility.

1 INTRODUCTION

The World Wide Web (often just referred to as "the web") represents a fertile area for computer simulation research. Combining the web with computer simulation can have a key impact on future simulation research. Among the directions one can take in this endeavor are (1) parallel and distributed model execution, and (2) distributed model repositories. Both these avenues are fruitful. We have narrowed our focus to the area of distributed model repositories since there has been less research in this area than in the more mature field of distributed simulation (Fujimoto, 1990; Lin and Fishwick, 1996). Also, the concept of model repository lends itself to the study of how to organize model information. Since the web is also concerned with how to effectively organize information, this appears to be a reasonable way to blend the web with simulation. The web defines a networked hypermedia approach to storing information. Search engines exist to help a user browse or perform a topical search. In simulation, information is generally focused on physical objects. These physical objects, whether they are humans, milling machines or a container of fluid, have attributes and exhibit behaviors. If we are to permit a situation where physical object information is as freely available as hypermedia to remote users on today's web, then we need to (1) formalize this information, (2) provide a way to integrate to today's web-based information, and (3) effect mechanisms for searching and browsing models. In this paper we explore these three issues in the context of OOPM and MOOSE.

MOOSE is an acronym for "Multimodel Object Oriented Simulation Environment", a modeling and simulation framework under development at University of Florida. MOOSE is an implementation of "Object Oriented Physical Modeling" (OOPM) (Fishwick 1996), which is an approach to modeling and simulation which defines a formal approach to capturing physical knowledge in a form that extends the object design principles specified in the fast-growing area of object design within software engineering and programming language design (Booch, 1994; Rumbaugh et al., 1991) Some of the current object-oriented design methodology requires modification to support physical modeling. Moreover, there does not cur-

rently exist a clearly-defined method of capturing physical knowledge in an object-oriented modeling framework even though many of the object-oriented "nuts and bolts" exist to help structure the method. The OOPM methodology satisfies the requirement of development of a theoretical framework for physical modeling, while allowing for legacy code insertion and user-defined dynamic model and multimodel types.

Initial development of MOOSE, focussed on an environment consisting of a single host system, has been completed, with results reported in detail by Cubert and Fishwick (1997a). The next step, now underway, involves expanding the environment to permit model definitions to be distributed over any number of hosts within the framework of the worldwide web. There are two kinds of distributed operation to consider: one is where model definitions are distributed, with some classes defined here, others 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 supports both kinds of distributed operation; with our emphasis being on distributing definition of multimodels.

We first briefly summarize some of MOOSE's focal ideas and properties, such as use of multimodels to facilitate model refinement to achieve appropriate levels of model fidelity, use of dynamic models, and reuse by design. Fuller treatment of these topics, as well as issues such as how MOOSE captures the geometry of a model, relation between conceptual model and simulation program, relations such as aggregation, containment, composition, usage, association, generalization, and specialization, validation and verification, extensibility, speed of development, and platforms and portability, have been addressed by the authors elsewhere (Cubert and Fishwick, 1997b). After presenting background on the components of MOOSE and how they interact, in sufficient detail to orient the reader, the major emphasis will focus on MOOSE Model Repository (MMR), because this is the vehicle which expands the horizons of MOOSE to the limits of the web.

Thus the balance of the paper is organized as follows. In Section 2 we briefly present focal issues such as multimodels, dynamic models, and reuse. Section 3 covers the components of MOOSE and how they interact. Section 4 goes into detail on MMR and distributed operation. Section 5 presents our conclusions and directions for future work.

2 MULTIMODELS, DYNAMIC MODELS, AND REUSE BY DESIGN

Derived from OOPM principles, MOOSE promises not only to tightly couple a model's human author into the modeling and simulation process through an intuitive human–computer interface (HCI), but also to help a model author to perform any or all of the following: (1) to think clearly about, to better understand, or to elucidate a model; (2) to participate in a collaborative modeling effort; (3) to repeatedly and painlessly refine a model as required, in order to achieve adequate fidelity at minimal development cost; (4) to painlessly build large models out of existing working smaller ones; (5) to start with a conceptual model which is intuitively clear to domain experts, and to unambiguously and automatically convert this to a simulation program; (6) to create or change a simulation program without being a programmer; (7) to perform simulation model execution and to present simulation results in a meaningful way so as to facilitate the other objectives above.

The degree of detail in a model reflects the model author's abstraction perspective (Fishwick, 1988). 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 (Fishwick 1989), and with certain objectives for the model or simulation to meet (Berzins 1986). 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 (Fishwick and Lee, 1996). 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), Equation Constraint Model (EQN), and Rule-based Model (RBM); 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 (Fishwick, 1995); additional dynamic model types will likely be added to the MOOSE repertoire; MOOSE has been designed to be extensible in this regard. In addition to model refinement during development, multimodeling may also be used during model execution: components of a multimodel may be behaviorally abstracted to fit time constraints placed upon model execution.

In MOOSE, dynamic behavior of the system is represented by *dynamic models*. Dynamic models are methods of the various classes in the conceptual

model. Dynamic models are readily added, changed, and removed, as part of model development, at any time. Here MOOSE makes good 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, EQN, and RBM, with others contemplated, such as Petri nets, and System Dynamics 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.

To support the kind of heterogeneous model hierarchies shown abstractly in Figure 1, we must ensure that our models are *closed under coupling*. In short, this suggests that the method of coupling one model component to another must be clearly defined. Two kinds of coupling exist: intralevel and interlevel. Intralevel coupling reflects model components coupled to one another in the same model. For example, one needs to specify rules of how Petri nets, compartmental models and System Dynamics graphs are formed. With a System Dynamics graph, a rule of model building defines that any level has an input rate and an output rate. A more interesting case arises in interlevel coupling since we must ensure that we define rules as to how model components from one model can be refined into models of different types. Can a finite state machine state be refined into a Petri net, or can a functional block model contain finite state machines (FSM) inside blocks? What are the rules to guide this refinement? The rule for intralevel coupling is based on functional composition. The primitive of *function* with its *input* and *output* defines the coupling procedure in the following way. All models are encapsulated in a single function. This represents the outer shell to support interlevel coupling. Within a model there are *functional entry points*. These are inner shells where new models may be optionally inserted. Each model type has its own entry point defined differently. For example, for the model type "FSM", we may define each state to be of the form: $v(state) = f()$ where $f()$ is an arbitrary function and $v(state)$ defines the value of the attribute *state*. If *state* is not refined, then $f()$ returns the value of the state as a character string or integer. If *state* is refined, then $f()$ may be replaced by *any* function—whether this function is a dynamic model or a code method. The coupling approaches are defined in more detail by Fishwick (1997).

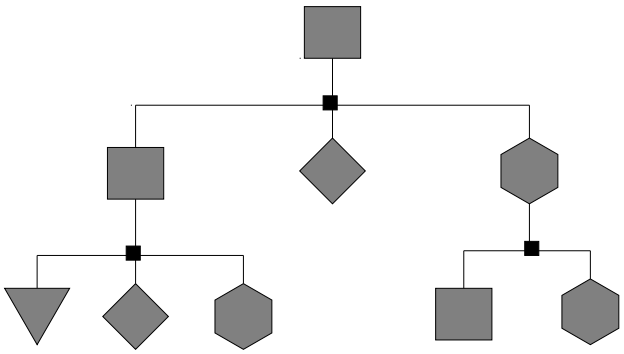

Figure 1: Multimodeling Tree Structure for Model Refinement; Polygons Depict the Heterogeneous Nature of Multimodeling: each type of Polygon represents one type of Dynamic Model

Reuse of one's own previous work, as well as by one model author of the work of others, is encouraged by availability of model repositories. An application framework such as MOOSE is more than just a class library. In an application framework, classes from the library are related in such a way that a class is not used in isolation but within a design encouraged and supported by the framework. The MOOSE Model Repository (MMR) is aptly named because it is not just a class library; as a model repository, it stores not only a collection of classes available for reuse, but also the design which relates those classes as to how they play together within the geometry and dynamics of a particular model. This enables support for one of Booch's (1994) five attributes of a complex system: "A complex system that works is invariably found to have evolved from a simpler system that worked .... A complex system designed from scratch never works and cannot be patched up to make it work.". Using MMR, model authors can start from some piece of their overall system that happens to appeal to them intuitively. When several such pieces are working, they may be combined into a more-complex (working) system.

3 COMPONENTS OF MOOSE

Components of MOOSE fall into three groups: Human Computer Interface (HCI), Library, and Back End. The HCI is comprised of two components: Modeler and Scenario. *Modeler* interacts with the human model author via graphical user interface (GUI) to construct the model. In simulation parlance, this is *model design*. Modeler relies on the Library (discussed below) to store model definitions. *Scenario* is

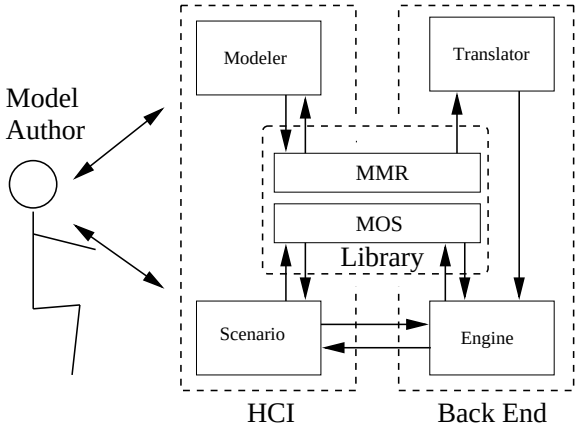

Figure 2: The three Components of MOOSE (HCI, Library, and Back End) shown outlined with dashed line Boxes; Parts within each Component are shown outlined with solid Boxes

a visualizer employing a GUI. Scenario activates and initializes simulation model execution (which we call Engine) at the behest of user (who may or may not be the original model author). Scenario maintains synchronous interaction with Engine, visualizing Engine output in a form meaningful to user, optionally allowing user to interact with Engine, including modifying simulation parameters and changing the rate of simulation progress.

Modeler GUI's "main" part defines classes and objects and relations among classes (aggregation and specialization or generalization) on one or more canvases. On the canvas, rectangles represent classes. These rectangles are joined by relations to form a tree, or, more generally, a graph, reflecting relations in the system being modeled. Some models look cleaner if aggregations and specializations are kept on separate canvases; this is supported but not required. Similarly, some models are large enough that several canvases are needed to capture the representation. Each class is a box which, when opened, reveals more information, and permits the model author to define the name of the class, its attributes, its methods, and its named objects. Within each method, the model author may specify input parameters and output parameters, as well as identifying which dynamic model type the method is to be. In addition to the "main" GUI presented above, there is a GUI editor for each dynamic model type, *i.e.*: the FSM editor for finite state machines, the FBM editor for functional block models, the EQN editor for equation constraint models, and the RBM editor for rule-based models.

The Back End has two components: Translator and Engine. *Translator* is a bridge between model design and model execution: Translator reads from the Library a language-neutral model definition produced by Modeler, and emits a complete computer program for the model, in Translator Target Language (TTL). Presently MOOSE TTL is C++; potentially, TTL can be Java or another language. This simulation program emitted by Translator in TTL is called *Engine.* Once compiled and linked with runtime support, the Engine executable is activated under control of Scenario to perform model execution. Library has two components: *MOOSE Model Repository (MMR)* and *MOOSE Object Store (MOS).* MOS holds object data and MMR holds object metadata. MMR keeps track of models as they are being built. MMR servers provide a database management system (DBMS) for model definitions. MMR clients work with Modeler and Translator to define and use model definitions. Models and model components created by other model authors (or the same model author previously) are available for browsing, inclusion, and/or reuse. Base classes such as sets for modeling collections and popular geometries for spatial models are available to the model author. An MMR client can simultaneously maintain conversations with several MMR servers on different hosts, thus permitting model definitions to be distributed. An MMR Server can simultaneously maintain conversations with several MMR clients, on the same or different hosts, which permits collaboration on model development. MOS does for objects much of what MMR does for models. MOS works with Engine and Scenario, in similar fashion to the way MMR works with Modeler and Translator. MOS manages object persistence. The architecture permits MOS to be capable of distributed operation, just like MMR, although this is not our focus in MOOSE.

4 MOOSE MODEL REPOSITORY (MMR) AND DISTRIBUTED OPERATION

There are two kinds of distributed operation to consider: one is where model definitions are distributed, with some classes defined here, others 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 supports both kinds of distributed operation; the present implementation supports distributing definition of multimodels, as this is our primary research focus.

The MMR originated in a perceived need which arose in the stand-alone version of MOOSE to unburden the Conceptual Modeler in the MOOSE HCI from maintaining complex structures and relations among classes, objects, attributes, methods, and parameters. Originally, the model definition provided as

output of the HCI was a set of flat text files, similar in some ways to the HTML (hypertext mark up language) now ubiquitous on the web. We had already developed in the MOOSE Translator a capability to read and parse this model definition and build the aforementioned structures and relations, so it was a relatively simple matter to reuse this code, and add a sockets-based (TCP) communications layer. This effort not only succeeded, it also paved the way for extending the horizon of MOOSE from stand-alone system to web operation. Along the way, we kept the flat file format we had designed, and thus preserved the capability to load the MMR from one or more sets of flat files describing any numbers of previously-constructed models. This capability now makes it easy for an MMR to import model definitions created or modified by hand, which are easy to handle, transmit, and modify because of the flat text file format. We back up the MMR into this format; we dump model definitions into this format. While the format can be readily machine generated, it is also amenable (just like HTML) to being edited by hand with one's favorite text editor.

The MMR has a client/server architecture, with each MMR server maintaining a database of model definitions. MMR is in some ways is patterned after the CORBA (Common Object Request Broker Architecture) IR (Interface Repository) (Orfali, 1996). MMR as a MOOSE component does its part to support an overall model/view architecture, with multiple views being possible for a single model, and similarly, with multiple models being present in a single view. In the original stand-alone mode, the clients were the MOOSE Conceptual Modeler and the MOOSE Translator, with the Conceptual Modeler updating the MMR server, and the Translator querying it in order to emit Engine code in TTL. There was one MMR server, and it was co-located on the same host with the two aforementioned clients. In web-wide operating mode, MMR servers can be anywhere, can exist in any number, and can be shared; if each host has an MMR server, then the system offers greatest robustness in the face of network outage, but the architecture does not require it. MMR clients will usually be MOOSE HCI's and Translators; several HCI's located far from one another may collaborate to share and reuse model components; or, several Translators located far apart and unaware of one another's existence can be interested in using the same model or component definition. However, it is also part of the plan to expose an interface for other clients, which may be programs of any kind, including perhaps web browsers or other programs. This open architecture invites use of distributed model definitions outside of MOOSE to broader realms, as a more general object-oriented application framework. Time will tell if this idea will gain acceptance. The underlying MMR design is independent of whether MOOSE operates in stand-alone mode, or with clients and servers in any number and located far apart.

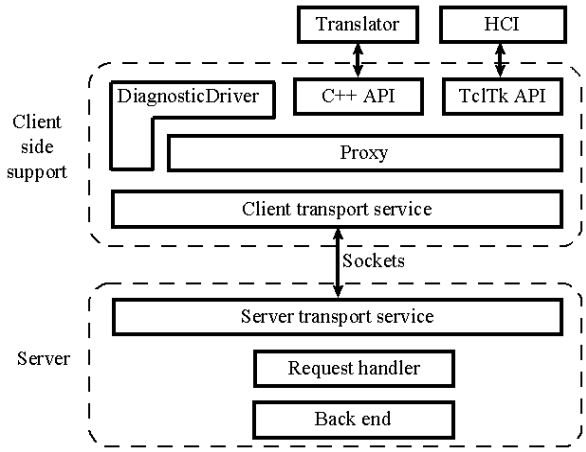

Figure 3: MOOSE Model Repository (MMR) Internals; Client above, Server below, each surrounded with dashed line; detail in accompanying text

We now examine the MMR architecture which appears in Figure 3. Clients communicate with MMR using client side support. Two API's are shown: one for C++ code and one for TclTk, which support our Translator and HCI, respectively. Other API's are possible, should support for client code in other languages be needed. The client side support is layered as shown. Presently, our client side support is relatively thin. The diagnostic driver, providing built in support for test and development, is GUI-based, and allows developers and system maintenance technicians to operate the interface to the MMR server without a client program, permitting tests of Proxy and Client transport layers of client side support, as well as all of the server. The client communicates with the server using sockets, which are supported in both our platforms of choice (Windows NT and Solaris Unix), enabling client and/or server to be positioned on either platform with complete transparency. Sockets work whether clients and server are located on the same or different hosts. Client transport service is written in TclTk in a style which applies the OO principles available (encapsulation and information hiding). Server transport service is written in C++ with class names such as *Sockets*, *Hosts*, and *Circuits*. Proxy's counterpart on the server side is Request handler. Proxy and Request Handler work together. To the extent that we want to stage or cache information on the client side, this is hidden within Proxy. As previously stated our intent is a thin

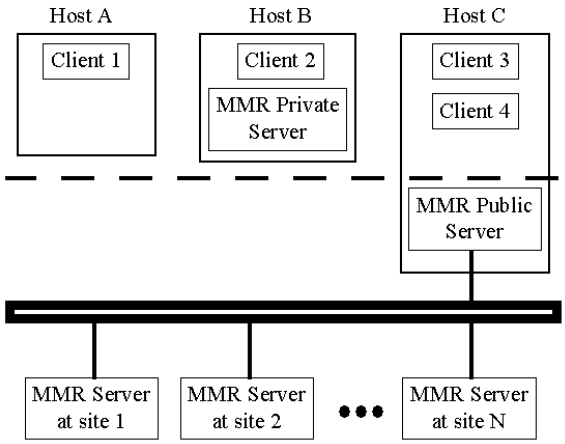

Figure 4: MOOSE Model Repository (MMR) as deployed on the Web; Dashed line is Firewall, above which is an Intranet; Heavy double line represents the Internet

client, but the presence of Proxy provides the ability to "thicken" the client side in the interest of performance, should that become necessary. Finally, the Back end provides data structures, linkage, and relations for classes, objects, attributes, methods, parameters, aggregation, association, containment, generalization, specialization, and inheritance; in short, the things one needs to know about a conceptual model.

Server Transport Service incorporates the initial sequence: create socket, set nonblocking, bind, and listen. Then periodically two activities are performed: accept'ing new connection requests, and servicing requests on existing connections, with priority given to the latter, and round-robin service policy. A dynamically-allocated self-expanding list of virtual circuits (connections) is maintained, so that an MMR Server can maintain any number of conversations with any number of clients and keep them all separate. Client Transport Service functions with send/receive pairs. Its receive is nonblocking; when there is no reply, the code is able to distinguish *end of file* from *no data yet*. This permits a client to retrieve long multi-message responses, and never to block. An interesting example of code reuse of the Client and Server Transport Services is this code also serves to synchronize Scenario and Engine, with Client Transport Service embedded into Scenario and Server Transport Service included in Engine runtime support.

Having examined MMR internal architecture, we now turn to two external views of MMR: first, the original stand-alone MOOSE which runs all processes on one host; second, distributed MOOSE which permits any number of MMR clients, any number of MMR servers, and located on an arbitrary collection of hosts. The first view appears in Figure 2, and is relatively simple, where the MMR clients are the (conceptual) Modeler and the Translator as discussed above. The second view appears in Figure 4, and to this we now turn our attention. Above the dashed line appear three hosts, connected in an intranet. The dashed line is a firewall. A random collection of four client applications are shown; typically, these are instanmces of MOOSE (Conceptual) Modeler and Translator. Also above the dashed line firewall is an MMR private server, which is accessible to all clients in the intranet but not to any clients outside (below the dashed line). Just below the dashed line firewall appears an MMR public server. This server is accessible not only to clients above the dashed line but also to clients throughout the web. The heavy double line represents the internet and TCP/IP MMR protocol. Several distant MMR servers are shown at various web sites. Specifically, suppose that Client 1 is an instance of (Conceptual) Modeler which is building a model some of whose components are stored locally, in either the private or public server, with other components located at the MMR at site 1 and at the MMR at site 2. Client 1 is able to construct its large model from the various small ones transparently with respect to the location of the components. The illusion that the model definition is all stored locally is maintained by cooperation among the MMR client-side support services attached to Client 1 on Host A, the MMR Public Server on Host C, and the (distant) MMR Servers at sites 1 and 2.

Present plans call for the MMR protocol to be identical to the format already in use for the stand-alone version of MOOSE; this format appears in the flat text files which describe MOOSE conceptual models, and is HTML-like in the sense that it can be inspected and modified with almost any text editor, to facilitate diagnostic work as well as customization. Since the web is a network of multimedia documents, we propose a way of integrating the MMR with the web. This is done by a simple mechanism: permitting an object attribute to be of type *URL*, a class whose instances are URL's. Thus, documentation is an attribute of an object and within a web document, a conceptual model may be inserted as a basic URL type, e.g., *model*. Accordingly, to retrieve a conceptual model of a six-cylinder automobile engine from Detroit, the following hypothetical URL would be accessed: *model://models.gm.com/eng6cyl.mod*. This permits a tightly-coupled, interwoven effect between the web and a MOOSE conceptual model. MMR Servers will support this proposed framework when it is available; in the interim, they can communicate with TCP/IP, until there is demand to implement the

proposed mechanism.

5 CONCLUSIONS AND FUTURE DIRECTIONS

To date MOOSE has fulfilled each promise we had for its capabilities. We are gratified that OOPM has provided both a sound theoretical footing as well as a guide for our intuition as we develop MOOSE. Several research projects (e.g., a study of the Everglades ecosystem) are planning to work with MOOSE, and this fall students will use MOOSE in homework and projects for the graduate course in Simulation Principles at University of Florida, providing what is certain to be valuable feedback.

Distributed web-based operation is leading in new directions. Distributed operation questions include (1) how to categorize and locate components for reuse, (2) whether dynamic binding is the most appropriate binding time for component definitions, (3) how scalable the MMR will turn out to be, (4) what relation if any will exist between MOOSE, CORBA, and DCOM, and (5) how successful will be our approach to embedding legacy code as MOOSE models. Other questions include (6) whether Java will displace TclTk as primary language for MOOSE HCI's, (7) how to apply distinctions with greater sophistication among the relations containment, usage, composition, and association, (8) how best to extend the existing MOOSE repertoire for dealing with collections of objects to make it better serve model authors' needs, and (9) how to make Scenario's visualizer as generic as the rest of the model definition.

ACKNOWLEDGMENTS

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 acknowledge with thanks the software development efforts of: Tolga Goktekin for the Conceptual Modeler, Youngsup Kim for the Functional Block Model Editor, and Gyooseok Kim for the Rule-Based Modeler. We are grateful to Kangsun Lee for assistance with model development, and to Jim Klosterboer of GRC International for providing some sockets code.

REFERENCES

Berzins, V., M. Gray, and D. Naumann. 1986. Abstraction Based Software Development. *Communications of the ACM*. 29 (5): 402–415.

Booch, G. 1994. *Object-Oriented Analysis and Design with applications*, 2nd ed. Reading, Massachusetts: Addison-Wesley.

Cubert, R. M. and P. A. Fishwick. 1997a. MOOSE: architecture of an object-oriented multimodeling simulation system. In *Proceedings of SPIE – Society of Photo-optical Instrumentation Engineers; Enabling Technology for Simulation Science*, ed. A. F. Sisti, 3083:78–88. Bellingham, Washington: Society of Photo-optical Instrumentation Engineers.

Cubert, R. M. and P. A. Fishwick. 1997b. MOOSE: an object-oriented multimodeling and simulation application framework. To appear in *Object-Oriented Application Frameworks*, ed. M. Fayad, D. Schmidt, and R. Johnson. New York : John Wiley and Sons.

Fishwick, P. A. 1988. The Role of Process Abstraction in Simulation. *IEEE Transactions on Systems, Man and Cybernetics*. 18 (1): 18–39.

Fishwick, P. A. 1989. Abstraction Level Traversal in Hierarchical Modeling. *Modeling and Simulation Methodology: Knowledge Systems Paradigms*, Zeigler, B. P., M. Elzas, and T. Oren, eds. Elsevier North Holland. 393–429.

Fishwick, P. A. 1995. *Simulation Model Design and Execution : Building Digital Worlds*. Englewood Cliffs, New Jersey: Prentice-Hall.

Fishwick, P. A. 1996. Extending Object Oriented Design for Physical Modeling. *ACM Transactions on Modeling and Computer Simulation*. Submitted July 1996.

Fishwick, P. A. and K. Lee. 1996. Two Methods for Exploiting Abstraction in Systems. *AI, Simulation and Planning in High Autonomy Systems*. 257–264.

Fishwick, P. A. 1997. A Visual Object-Oriented Multimodeling Design Approach for Physical Modeling. Submitted April 1997 to *ACM Transactions on Modeling and Computer Simulation*.

Fujimoto, R. M. 1990. Parallel Discrete Event Simulation. *Communications of the ACM*. 33 (10):31–53.

Lin, Y. B. and P. A. Fishwick. 1996. Asynchronous Parallel Discrete Event Simulation. *IEEE Transactions on Systems, Man and Cybernetics*. 26 (4):397–412.

Orfali, R., D. Harkey, and J. Edwards. 1996. *The Essential Distributed Objects Survival Guide*. New York : John Wiley and Sons.

Rumbaugh, J., M. Blaha, W. Premerlani, E. Frederick, and W. Lorenson. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall.

AUTHOR BIOGRAPHIES

ROBERT M. CUBERT is a Ph.D. student in the Department of Computer and Information Science and Engineering at University of Florida. His research interest is distributed object-oriented modeling and 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 communications.

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.