INTEGRATING CONTINUOUS AND DISCRETE MODELS WITH OBJECT ORIENTED PHYSICAL MODELING

Paul A. Fishwick
Dept. of Computer & Information Science and Engineering
University of Florida
Bldg. CSE, Room 301
Gainesville, FL 32611

ABSTRACT

When we build simulation models and construct dynamical models for physical systems, we often do not do so using a clear overall framework that organizes our geometry, dynamics and models. How do geometry and dynamics intertwine to effect system change over multiple abstraction levels? We present a methodology, called *object-oriented physical modeling* (OOPM), which builds on the currently accepted computer science approach in object-oriented program design. This type of modeling injects a way of incorporating geometry and dynamics into general object-oriented design. Moreover, we present an approach to dynamical modeling that mirrors major categories of computer programming languages, thereby achieving a definition of system modeling that reinforces the relation of *model* to *program*.

1 INTRODUCTION

Simulation is divided into three areas: 1) *model design*, 2) *model execution*, and 3) *execution analysis*. Modeling is the process of abstracting real world behavior into a more economical form for purposes of experimentation and learning. Our chief interest is in efficiently capturing and organizing the knowledge necessary to simulate physical systems, both artificial and natural. Simulation requires that we have a way of designing and executing models. Models can represent geometric shapes of real objects or the dynamics of those objects. The shape of an object is captured by its geometry, which is used by computer graphics to display the object. The dynamics of an object allows us to view a computer animation of the object undergoing time-dependent change. Our purpose is to specify a method for modeling a physical system, while claiming that our method provides benefits such as model and model component reuse. We provide a way of organizing physical knowledge and a methodology for those wanting to model systems at many levels of detail. Our techniques build on top of object-oriented design principles espoused in both computer science as well as computer simulation. The method surfaces the importance of an integrated object-process method where both *objects* and *processes* are made visible in model design. We present a uniform method that extends previous work by specifying how shape and dynamics are integrated in the same framework, and by defining methods for modeling that permit an integration of existing modeling methods without recommending an all-encompassing new modeling technique. In this sense, the contribution is one where we devise a specific *model integration* approach. Further work in a more complete OOPM methodology can be found in (Fishwick 1996a)[1].

2 MOTIVATION AND BACKGROUND

The word *model* is a somewhat overloaded term and can have many meanings depending on context. We proceed to define what we mean by the word *model*. Models are devices used by scientists and engineers to communicate with one another using a concise—often visual—representation of a physical system. Models are visual high-level constructs that we use to communicate system dynamics without the need for frequent communication of low-level formalism, semantics and computer code. In our methodology (Fishwick 1995), a *model* is defined as one of the following: 1) a graph consisting of nodes, arcs and labels, 2) a set of rules, or 3) a set of equations. Computer code and programs are not considered to be models since code semantics are specified at too low a level. Likewise, formal methods (Padulo and Arbib 1974; Zeigler 1989) associate the formal semantics with models but do not focus on representing the kind of high-level form needed for modeling. One of our thrusts in this paper is to discourage readers from thinking that to simulate, they need to choose a pro-

[1] http://www.cise.ufl.edu/~fishwick/tr/tr96-026.html

gramming language and then proceed directly to the coding phase. Even the phrase "object-oriented" is often defined as being synonymous with certain programming languages such as C++, and so one may be lead to begin programming, using polymorphism, virtual base classes and other artifacts, before the design is specified. Without the proper scaffolding for our models, in the form of a conceptual model, we will produce disorganized pieces of code without a good understanding and organization of the physical process we wish to study. The act of coding in an object-oriented language is not a substitute for doing good design. As an example, C++ provides many object oriented capabilities, but does not enforce object oriented design. Norman (1988) points out the need for good visual, conceptual models in general design for improved user-interfaces to physical instruments and devices. The importance of design extends to all scientific endeavors with a focus on models. Models need to provide a *map* between the physical world and what we wish to design and subsequently implement either as a program or a physical construction.

Programs and formal specifications (Zeigler 1972; Zeigler 1989; Praehofer 1991) are a vital ingredient in the simulation process since, without these methods, modeling approaches lack precision and cohesion. However, formal specifications should not take the place of models since they serve two different purposes. Specifications are needed to disambiguate the semantics, at the lowest level, of what one is modeling. Models exist to allow humans to communicate about the dynamics and geometry of real world objects. Our definition of modeling is described at a level where models are translated into executable programs and formal specifications. Fishwick and Zeigler (1992) demonstrated this translation using the DEVS (Zeigler 1989) formalism for one particular type of visual multimodel (finite state machine model controlling a set of constraint models). For other types of multimodels, one can devise additional formalisms (Praehofer 1991). Object-oriented methodology in simulation has a long history, as with the introduction of the Simula language (Birtwistle 1979), which can be considered one of the pioneering ways in which simulation applied itself to "object-oriented thinking." Simula provided many of the basic primitives for class construction and object oriented principles but was not accompanied by a visually-oriented engineering approach to model building that is found in more recent texts (Rumbaugh et al. 1991; Booch 1991; Graham 1991). As we shall see in model design, the visual orientation is critical since it represents the way most scientists and engineers reason about physical problems and, therefore, must be made *explicit* in modeling. Other more recent simulation thrusts in the object-oriented arena include SCS conferences (Roberts, Beaumariage, Herring, and Wallace 1995) as well as numerous Winter Simulation Conference sessions over the past ten years. Also, various simulation groups have adopted the general object-oriented perspective (Rothenberg 1989; Zeigler 1990; Balci and Nance 1987).

We specify two contributions: 1) a comprehensive methodology for constructing physical objects that encapsulate both geometric and dynamical models, and 2) a new taxonomy for dynamic models. The motivation for the first contribution is that there currently exists no method that uses object-oriented design and specifies an enhancement of this design to accommodate static and dynamic models. We have taken the existing visual object-oriented design approaches reflected in texts such as Rumbaugh (1991) and Booch (Booch 1990) and extended these approaches.

3 SCENARIO

A general scenario should be defined so we can develop the concepts of physically-based object-oriented design. We will create a simple example and then provide a table that illustrates how this example can be seen for a wide variety of disciplines. The reason for this choice of scenario is that it captures the essence of physical modeling: the application of dynamics and geometry using particles and fields. Even in the domain of sub-atomic particles, the object orientation is relevant since particles and fields integrate to form objects via Schroedinger's wave equation. The robot serves the role of a particle and the space (or landscape) serves the role of space. Together, they provide for a comprehensive model. Consider a 2D space s that is partitioned using either a quadtree or array. A set of mobile robots move around s, undergoing change, as well as changing attributes of s. Fig. 1 illustrates s and a sample robot r. In the remainder of the paper, we will refer to Fig. 1 using classes, objects, attributes and methods. A robot or set of robots move around space s, some staying within the confines of a particular partition of s, such as $s_{i,j}$ for $i, j \in \{1, \ldots, 8\}$. Moreover, certain attributes of s may change. For example, there may be water in s or a particular density of matter assigned to s. Our robots will be unusual in that they are capable of changing shape over time if the dynamics demand this of them. Before we embark on a discussion of object-oriented physical design for robots within spaces, we present Table 1 to illustrate how, through mappings from one discipline to another, different areas fit into this general scenario scheme.

Table 1: A sample set of applications using a particle-field metaphor.

Application	Particle	Field
Cybernetics (intelligent agents)	robot	space (room,factory floor, terrain)
Military (Air Force)	plane, squadron	air space
Ecology	individuals, species	landscape
Materials	particles,molecules	fluid (air, liquid)
Computer Engineering	chip, module	N/A
Quantum Mechanics	wave function	wave function
Meteorology	hurricane, tornado (finite volumes)	atmosphere

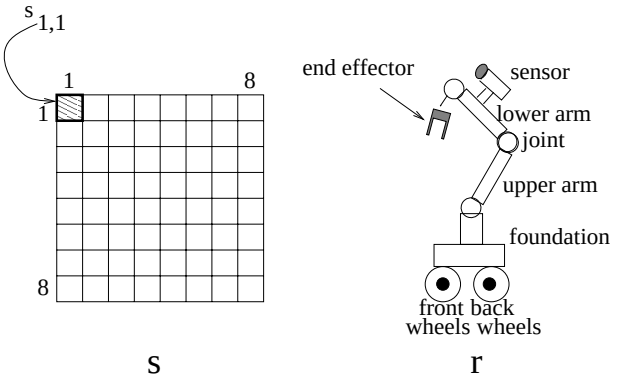

Figure 1: Scenario for generic object behavior.

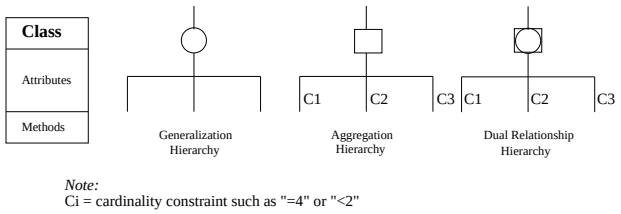

Figure 2: Structure of a class with three relations.

4 CONCEPTUAL MODELING

The first phase is constructing a conceptual model of the physical scenario. To build such a model, we must construct a class graph with relations among the classes. Furthermore, we must identify attributes and methods in those classes. A class is a type. Our treatment of a class is as if it served as a "cookie cutter." A cookie cutter (class) operates over a sheet of dough to create cookies (objects). We might create several types of robots: *Walking*, *Rotating*, *Fixed-base*. Each of these are classes and they are sub-classes of robot since all of them are types of robots. This particular relation is called *generalization*. Another kind of useful relation is called *aggregation* since it involves a relation among classes where there is a "part of" relationship. For example, a particular robot may be composed of (or aggregated from) wheels, an arm and a camera. The base, arm and camera are part of the robot: the robot is an aggregate of the base, arm and camera. Fig. 2 shows how we illustrate both of these relations: generalization with a circle and aggregation with a square. We also permit an analyst to specify any given relation as both aggregation and generalization. This is delineated with a circle inside a square. The C specified in Fig. 2 can specify cardinality for a class. In general, without such a specification, it is assumed that a class can be composed of any number of objects of the sub-class. However, let's say that we create a class *Room* which will always have four walls, then we can specify $= 4$ on the aggregation relation arc to illustrate this constraint. Without this explicit constraint, a *Room* can be composed of any number of walls. This approach is consistent with several existing OO approaches (Rumbaugh et al. 1991; Hill 1996) to aggregation specification.

While we are on the subject of classes, we define an object to be an instance of a class. A particular *wheel* is an instance of the class called *Wheel*. A class is a set of objects, which are related through the class definition. A class is composed of its name, a set of attributes and a set of methods. Aggregation among classes requires some clarification. If The set

of *Wheels* for a robot is aggregation from the classes *FrontW* and *BackW*, there may be any number of front wheels and any number of back wheels. The aggregation just shows the class aggregation and not the object aggregation. The specific number of wheels is something that is changed as we create instances of the two classes. An actual object called *wheels* can be created from class *Wheels* and then we can create two objects from *FrontW* and two from *BackW*. We can even create a containment model (or data structure) which we locate as an attribute value within *wheels* that shows the composition of this particular *wheels* object.

A key part of conceptual modeling is identifying the classes. For the most part, this procedure is ill-defined but some rules and approaches do exist (Fishwick 1995; Graham 1991) to help in the model engineering process. Natural language provides one basis on which to base choices for classes, attributes and methods. The following are heuristics to aid in the creation of the conceptual model from a textual description of a physical scenario:

- *Make nouns classes or instances of a class.*
- *Use adjectives to make class attributes, subclasses or instances.*
- *Make transitive verbs methods which respond to inputs.*
- *Use intransitive verbs to specify attributes.*

In the physical sciences and engineering, we use models to describe the shape of objects as well as their behavior. We call the combination of attributes and methods *structure*. The two types of relations among classes, *generalization* and *aggregation*, are very popular and are frequently used in object-oriented design. The reason for their utility and popularity is that they involve the implicit act of passing structure from one class to another. Structure passing is powerful and enables us to fragment the world into classes, while designing common structure through aggregation and generalization relations.

The passing of structure for generalization is top to bottom, of a hierarchy of classes related via generalization, and is frequently known as inheritance. Let's consider an orange. An *Orange* class inherits certain attributes and methods (i.e., structure) from the *Fruit* class since an orange is a kind of fruit. A walking robot is a type of general robot, and so inherits structure from the general robot. The uppermost classes in a generalization hierarchy are *base* classes and the child classes are *derived* classes from the particular base class.

We have discussed generalization and its associated structure-passing inheritance capability, but there is another key kind of relation: aggregation. Aggregation enjoys the benefit of structure passing also, but the structure passing in aggregation is bottom-up instead of top-down. A class that is an aggregation of classes underneath it captures the structure of all of its children. A robot contains all attributes and methods associated with each of its sub-components, such as links, cameras and the behaviors of those sub-components. As we use *inheritance* for generalization, we will use *composition* for aggregation. Structure passing is done, therefore, through inheritance and composition. For our discussion of generalization and aggregation, we assume that structure passing is a *logical* operation; however, it may not be directly implemented in a specific programming language or implementation. For example, a large 10^6 square cell space is an aggregate of 10^6 individual cells; an implementation may choose not to cause the explicit passing of structure from children (i.e., cell) to parent (i.e., space), nevertheless, the structure passing is a logical consequence of aggregation, and is *logically* present in our design, if not in our implementation.

Inheritance and composition are further defined as follows:

- Inheritance (or generalization) is the relational property of a generalization hierarchy. Composition (or aggregation) is the relational property of an aggregation hierarchy.
- To differentiate between layers in a hierarchy we use the terms "child" and "parent." A parent is always above a child regardless of the relation type. Therefore a child class in generalization inherits from its parent, but a parent class aggregates from its children.
- The words "derived" and "base" are *relative* to the type of relation. Base classes in a generalization tree are at the top with lower-level classes deriving structure. For aggregation, it is the opposite, with the base classes being at the leaves of the tree. Structure passing for both relations is derived from "base" to "derived" classes.
- The only classes that can be used for constructing objects are the leaf classes of a generalization hierarchy. Internal tree nodes are used to hold class structure but are not used for object construction.
- Inheritance occurs when a derived leaf class in a generalization class hierarchy is used to construct or create an object. The object "inher-

its" all attributes and methods from its parent (or *parents* in multiple inheritance).

- Composition occurs when any class in an aggregation class hierarchy is used to construct or create an object. The object passes all structure assigned to it upward to the parent (or *parents*) in multiple composition). The derivation of structure moves from the base classes.

Aggregation and containment are two different, but related concepts. Aggregation involves composition, which means that a class is composed of sub-classes. A glass of marbles will contain marbles, but the glass is not composed of marbles and so a *Marble* class, in an aggregation sense, is not a sub-class of *Glass* unless one redefines the meaning of *Glass* to encompass not only the physical structure of a glass, but also of everything within the "scope" or "environment" of the glass.

Generalization and aggregation are the key relations used in building a conceptual model, but they are not the only types of relations. If our physical scenario was such that all robots were attached to wooden boards, then one could form a relation arc between the class robot and the class board. However, there are sometimes better mechanisms for handling such cases. For the robot and the board, one can form an aggregate class called *Robot-env* which aggregates both board and robot classes. Some of these other class relations may or may not involve structure passing, but generalization and aggregation represent the power of a transitive stucture-passing relation involving any number of hierarchical levels. In any event, by allowing arbitrary relations among classes, we generalize conceptual models to have similar capabilities in representation to that of semantic networks and certain schemata in databases. That is, one can use logical inference and querying on the conceptual model in addition to using it only for structure passing. Also, the conceptual model need not be static. The conceptual model as it is originally defined represents a physical system at an initial time instant. New classes and relations may be added over time to permit a dynamically changing physical environment.

Fig. 2 illustrates generalization (◯) and aggregation (□) relations. It is not necessary to group all relations into one graph or hierarchy—multiple graphs or hierarchies are possible. Fig. 1 provides us with the base classes for Fig. 3. The downward and upward block arrows in Fig. 3 illustrates the respective directions of structure passing for inheritance and composition.

Inheritance and aggregation have *rules* that they use to perform the movement of structure within a tree. For inheritance, methods and attributes are

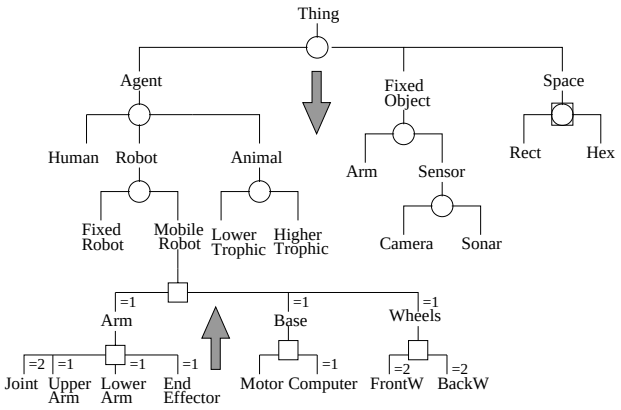

Figure 3: Phase 1, Step 1: Identify classes and relations.

copied from the tree root to the tree leaves except for when one overrides an inherited attribute or method. However, we need to be explicit about our "inheritance rule" since copies can be made, potentially, not only from the root of the generalization tree, or from the immediate parent of a class, but also from any class which lay in-between the root and leaf class. An attribute *type* within class *Agent* can be set to "organic". While the classes *Human* and *Animal* automatically inherit this attribute from *Agent*, *Robot* overrides this by setting *type* to "artificial." Overriding method and attributes permits a kind of heterogeneity in the derived classes so that they need not all be perfect copies of the base class. In addition to overriding, some structure from the base class may not be available for copying to derived classes. A default inheritance rule is one where a derived class inherits structure through *copying* from its parent class.

For aggregation, we have a more complex situation where an aggregation procedure must be specified for all attributes and methods in the base classes. An example of the need for such procedures is when two base-class methods or attributes are identical in name. The aggregation question is framed as "Given a number of base classes, how do we glue the base class attributes and methods to create attributes and methods in the aggregate class?" There is a conflict and a resolution method is required. Consider attribute *contains* in *Arm*, *Base* and *Wheels*. The *contains* attribute points to a data structure of what is contained within an object. Through composition, *Arm* obtains all three of these *contains* structures but what is *Arm* to do with them since they are all of the same attribute name? A logical aggregation procedure here is to say that all sub-classes of *Arm* with a *contains* attribute are grouped together into a record or array which is then placed in *Arm*. However, this sort of aggregation is not always appropriate. If the

sub-classes contain an attribute *count* which specifies how many objects there are of this class, then the correct aggregation rule for *count* within *Arm* involves a sum of all *count* attributes in the sub-classes of *Arm*. In aggregation, some sort of "aggregation rule" is *always* necessary. Implicitly, one could define that attributes of different names simply agglomerate into aggregate objects in a set-union fashion. However, there are many instances where this is not so. The *count* attribute is just one example. Other examples include aggregation into a matrix or array, and aggregation via model component "coupling" composing a dynamic model of sub-object methods. Rules can be stored in their respective classes. We make no attempt to formalize the rule structure—only to state that some code or rules should be available within a class to handle all aggregations that occur. A default aggregation rule is one where a derived class aggregates structure through *set union* of the child classes. That is—they just collect structure together without resolving conflicts, merging structure through summation or integration, or performing concatenation of structure.

As to how we might refer to populations or groups versus individuals, we consider the motor example. The set of motors can be called *motor* which points to a data structure specifying motor objects, while an individual motor requires an index such as *motor[2]*. When an object is created that uses the same root name for an object that already exists, such as when one created object *motor[2]* after having created *motor*, then the a hierarchy is assumed and aggregation occurs as a result. This mechanism allows one to attach recursive, hierarchical properties to any class in the conceptual model without explicitly specifying these properties at the time of conceptual model formation. Instead, this sort of multi-level hierarchy is defined as a static model. An example of this can be seen with the class *Rect* in Fig. 3. This class can be used to create a rectangular space hierarchy of any dimension. Let's first create a space object called *cell* by defining *Rect cell*. If we wish to structure this space into a quadtree, for instance, we can then create four new objects *cell[0], cell[1], cell[2]* and *cell[3]*. Since we used the same name *cell*, there is an automatic aggregation relation with *cell* composed of *cell[0], cell[1], cell[2]* and *cell[3]*. Specifically, aggregation rules come into play as previously described. The actual quadtree would be stored as a static model of *cell*. The explicit creation of aggregation hierarchies within the conceptual model is dictated by a heterogeneous aggregation relation. If a robot arm consists of exactly two links, which are different in nature, then this aggregation relation belongs in the conceptual model. However, the potentially infinite recursion of spatial decomposition suggests a homogeneous aggregrate relation, and is relegated to a static model stored within a "space object."

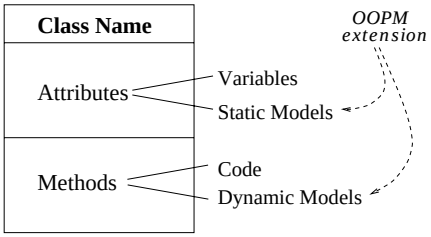

Figure 4: Structure of a Class.

OOPM specifies that an attribute is one of two types: variable or *static model*. Likewise, a method of one of two types: code or *dynamic model*. A method can be of a functional (representing a function) or constraint (representing a relation) nature. Once the conceptual model has been constructed, we identify the attributes and methods for each class. An attribute is a *variable*, whose value is one of the common data types—or a *static model*. A method can be *code*, whose form depends on the programming language, or a *dynamic model*. The structure of a class is seen in Fig. 4. Variables and code are described in OO languages such as C++ (Stroustrup 1991). We define a static model as a graph of objects and a dynamic model as a graph of attributes and methods. The model types of interest here are dynamic. However, the concept of static model complements the concept of dynamic model: methods operate on attributes to effect change in an object. Dynamic models operate on static models and variable attributes to effect change. We will use the following notation in discussing object-oriented terms. When we speak of a class, we capitalize the first letter, as in *Robot* or *Arm*. An object is lower case. An attribute that is a variable is lower case, whereas an attribute that is a static model is upper case for the first letter. A similar convention is followed for methods: a code method uses lower case with a parentheses "()" as a suffix; a dynamic model method is the same but with a capitalized first letter. Classes are separated from objects with a double colon "::" whereas objects are separated from attributes and methods using a period "." This convention is similar to the C++ language and is a convenience when communicating conceptual models textually. All classes, objects, attributes, and methods will use an italics font to differentiate them from surrounding text.

Fig. 5 displays another robot-oriented conceptual model to illustrate some of the points we've made about generalization and abstraction. We use the following new acronyms: *DigitalTech* for "digital tech-

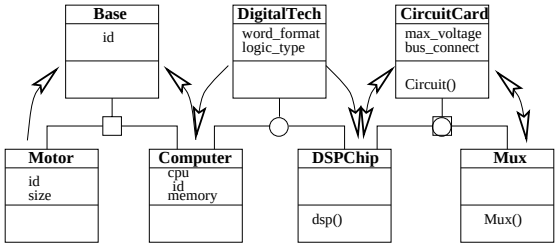

Figure 5: Generic generalization and aggregation scenario.

nology," *DSPChip* for "digital signal processing chip," and *Mux* for multiplexer. For each relation, we need to have a procedure. We'll proceed from left to right.

- Relation 1 (Aggregation):
  1. By default, *Base* will aggregate all methods and attributes using the set operation *union* ($\cup$) unless otherwise specified. We'll use this default to pass up all attributes of *Motor* and *Computer* except for *id*.
  2. *Base.id* is formed by *Motor.id* and *Computer.id* by concatenating them in a vector [Base.id,Motor.id].
- Relation 2 (Generalization):
  1. By default, *Computer* and *DSPChip* inherit all structure from *DigitalTech* through copying. We'll adopt the default on this one.
- Relation 3 (Dual):
  1. The default structure passing for this is the same as for both aggregation and generalization. We are stating that a *CircuitCard* serves as a composition of *DSPChip* and *Mux* as well as stating that *DSPChip* and *Mux* are *types* of *CircuitCard*.
  2. For aggregation, we specify model *Circuit()* as being defined by a functional coupling of *dsp()* and *Mux()*.
  3. For generalization, we use the default for inheriting *max_voltage*, but override the inheritance for *bus_connect* since the bus connection is an attribute of the circuit card and not relevant to *DSPChip* or *Mux*.

We've seen that generalization and aggregation provide us with power for structure passing, but that we require both default procedures as well as special procedures which either limit the structure passing in a particular way or accurately define it.

5 CONCLUSIONS

To build simple systems, we may sometimes get away without using a model design. In such a case, we may sketch a few formulae and proceed directly to the coding phase. However, with the increasing speed of personal computers, we are in a period of increased development for model design that might best be captured by the word "integration." As scientists and engineers, we have our own individual static and dynamic models for our part of the world. But this this not enough when we want to integrate models together. Suddenly, we find ourselves overwhelmed with the sheer size of model types, and frequently some may not have model types but have only coded their simulations. To get a handle on this situation, we need a blueprint as if we were going to perform this integration as a metaphor to building a house. Without a blueprint, the electrician and carpenter are at odds as to how to interact. They each construct their own complex parts and one only hopes that the resulting glued-together construction will function as a whole. The blueprint helps them to work together. Our methodology for object-oriented physical design is like the blueprint, permitting models of different types to fit together so that more complex and larger systems can be studied. These larger systems require an interdisciplinary approach to model design and so we must agree on a basic language for blueprints.

Our immediate goals are to apply this general methodology to various technical areas including the simulation of multi-phase particle flows in the University of Florida Engineering Research Center for Particle Science and Technology and an integrated modeling environment for studying the effects of changes in hydrology to the Everglades ecosystem managed by the South Florida Water Management District. For decision making in the military, many levels of command and control exist, and the methodology provides a consistent approach in using models for planning and mission analysis both "before action" and during "after action review." All of these systems have a characteristic in common even though they may appear at first quite different: they involve the modeling of highly complex, multi-level environments, often with individual code and models developed by different people from different disciplines. MOOSE development is underway and C++ code and GUI interfaces are being constructed to make it possible for analysts to use our system. A longer range goal is to allow our models to be distributed over the Internet (or over processors for a parallel machine). The object oriented concepts of re-use and encapsulation will help greatly in this endeavor. Also, we are trying to create a bridge between the use of

modeling in simulation and general purpose programming. As various authors have noted (Budd 1991; Fishwick 1996b), if one liberally applies the concept of metaphor to software engineering, the differences between software and systems engineering begin to dwindle to the point where software can be considered a *modeling process*.

ACKNOWLEDGMENTS

I would like to acknowledge the graduate students of the MOOSE team for their individual efforts in making MOOSE a reality: Robert Cubert, Tolga Goktekin, Gyooseok Kim, Jin Joo Lee, Kangsun Lee, and Brian Thorndyke. 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.

REFERENCES

Balci, O. and R. E. Nance. 1987. Simulation Model Development Environments: A Research Prototype. *Journal of the Operational Research Society 38*(8), 753 – 763.

Birtwistle, G. M. 1979. *Discrete Event Modelling on SIMULA*. Macmillan.

Booch, G. 1990. On the Concepts of Object-Oriented Design. In P. A. Ng and R. T. Yeh (Eds.), *Modern Software Engineering*, Chapter 6, pp. 165 – 204. Van Nostrand Reinhold.

Booch, G. 1991. *Object Oriented Design*. Benjamin Cummings.

Budd, T. 1991. *An Introduction to Object Oriented Programming*. Addison-Wesley.

Fishwick, P. A. 1995. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall.

Fishwick, P. A. 1996a. Extending object oriented design for physical modeling. *ACM Transactions on Modeling and Computer Simulation*. Submitted for review.

Fishwick, P. A. 1996b. Toward a Convergence of Systems and Software Engineering. *IEEE Transactions on Systems, Man and Cybernetics*. Submitted for review.

Fishwick, P. A. and B. P. Zeigler. 1992. A Multimodel Methodology for Qualitative Model Engineering. *ACM Transactions on Modeling and Computer Simulation 2*(1), 52–81.

Graham, I. 1991. *Object Oriented Methods*. Addison Wesley.

Hill, D. R. C. 1996. *Object-Oriented Analysis and Simulation*. Addison-Wesley.

Norman, D. A. 1988. *The Design of Everyday Things*. New York: Currency Doubleday.

Padulo, L. and M. A. Arbib. 1974. *Systems Theory: A Unified State Space Approach to Continuous and Discrete Systems*. Philadelphia, PA: W. B. Saunders.

Praehofer, H. 1991. Systems Theoretic Formalisms for Combined Discrete-Continuous System Simulation. *International Journal of General Systems 19*(3), 219–240.

Roberts, C. A., T. Beaumariage, C. Herring, and J. Wallace. 1995. *Object Oriented Simulation*. Society for Computer Simulation International.

Rothenberg, J. 1989. Object-Oriented Simulation: Where do we go from here? Technical report, RAND Corporation.

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

Stroustrup, B. 1991. *The C++ Programming Language* (2 ed.). Addison Wesley.

Zeigler, B. P. 1972. Towards a Formal Theory of Modelling and Simulation: Structure Preserving Morphisms. *Journal of the Association for Computing Machinery 19*(4), 742 – 764.

Zeigler, B. P. 1989. DEVS Representation of Dynamical Systems: Event-Based Intelligent Control. *Proceedings of the IEEE 77*(1), 72 – 80.

Zeigler, B. P. 1990. *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. Academic Press.

BIOGRAPHY

Paul A. Fishwick is an Associate Professor in the Department of Computer and Information Science and Engineering at the University of Florida. He received the 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. A complete biography and resume can be found at Dr. Fishwick's WWW home page: `http://www.cise.ufl.edu/~fishwick`.