Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

AN ARCHITECTURAL DESIGN FOR DIGITAL OBJECTS

Paul A. Fishwick

Department of Computer & Information Science & Engineering
University of Florida
Gainesville, Florida 32611, U.S.A.

ABSTRACT

We define the term "digital object" and specify a variety of qualities that are important during the object design phase. A digital object contains a set of models, and is meant to serve as a reusable entity to be used on the web or over the Internet. An example using a two link robot arm is presented. We have found the digital object design methodology to provide an information schema to describe where to locate information about the object, for simulation of dynamic models as well as the execution of other model types.

1 INTRODUCTION

For simulation to work effectively, we must have a collaborative, shared information architecture or repository where we can represent models. Moreover, we would like this schema to be as general as possible, admitting all types of simulation, and all form of model. While this may seem to be an horribly complicated task, we have created a set of guidelines for creating such a schema. The primary goals of this architecture are to 1) allow the scientist or engineer to reason about physical objects and their interactions, 2) provide a clear link between what is done on the computer (i.e., programming) and what is done by the analyst (i.e., modeling), and to suggest the foundations for a future standard on digital object reusability. We term the architecture Object Oriented Physical Modeling (OOPM) (Fishwick 1996) since all information is materially grounded on the physical structure under investigation.

We will employ the example of a two link robot arm attached to the ground, shown in Figure 1. We use a convention of using a lower case first letter for representing physical objects, and leave the upper case situation for representing classes (ref. 4.4). The entire scenario s contains a two link robot r. Link $l1$ contains joint $j1$ and arm $a1$, and link $l2$ contains joint $j2$ and arm $a2$. The joints are revolute and allow the arm to move in the represented two-dimensional plane. The end effector e is able to grasp onto objects. The two arms have frames of reference $f1$ and $f2$, which are considered to be part of their geometry models. The ground g supports r and has its own frame fg.

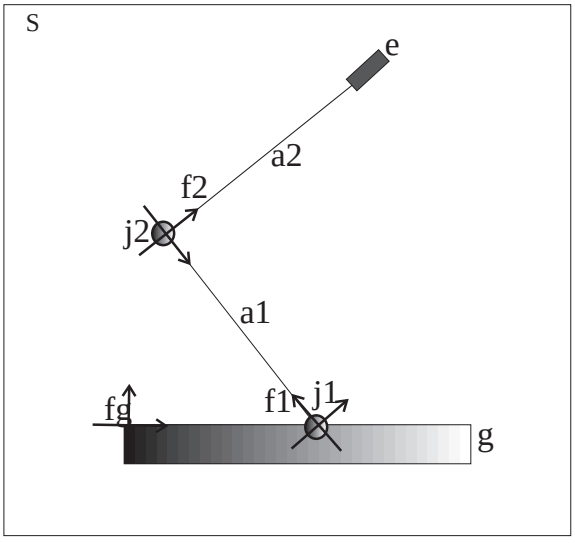

Figure 1: A two-link planar robot.

2 OBJECT PARADIGM

If we are to capture models inside objects, we need some sort of information paradigm. The object-oriented approach has gained favor for software engineering projects, and is also useful for thinking of physical objects and their components. We can think of all physical (where physical means "material") objects as being represented by digital objects inside the computer. Using the physical metaphor also falls under this scope, where a non-physical item such as a "queue," for example, could easily be turned into a set of physical objects. The digital object contains a set of

models as attributes. In addition to having attributes, an object can have methods; however, we often bind methods to models, which are encapsulated within their own objects. The use of objects is highly intuitive since our natural language is founded on object names and properties. We will always defer to the way in which physical objects are created, destroyed and modified as to how digital objects are to be represented. In the natural world, objects change over time and objects can be created from other objects, and so we should maintain these capabilities in the corresponding digital world.

The word "model" is severly overloaded, and yet we talk of *this model* or *that model* whenever we speak of phenomena. Let's define model as an abstract form of representation that substitutes for the physical phenomenon. Example model types are *dynamic* and *geometric*. The dynamic model serves to represent the behavior of a physical object, and the geometric model represents the object's structure or shape. Each model type abstractly captures some aspect of the real object, and so the digital object is simply a collection of all of these models. Models tend to be visual because visualization of phenomena aids in our understanding of mechanism. All modern simulation software packages are driven by graphical user interfaces. The user interface and the model are intertwined and can be considered identical; the model is the window to the physical object. Some models may be textual, and some analysts may prefer to think of programming language source lines as model components. Intrinsically, there is nothing wrong with this opinion, however in practice, most analysts will agree that "model" and "code" are different. In particular, code is an end product of having translated the model. Therefore, the code and the model are at two different abstraction levels, and therein lies the difference: level of translation or abstraction. A similar argument can be made for formal languages representing the semantics of continuous and discrete event phenomena. It is not that these languages compete with more abstract, often visual, representations. Instead, we form a chain of translation from representations that are easy to understand to lower level representations that maintain their own particular benefits. We are working with Zeigler's group on unifying OOPM and DEVS methodologies (Barros, Zeigler, and Fishwick 1998). OOPM currently has an implementation (Cubert and Fishwick 1997) which uses C++ as its target language, whereas (Barros, Zeigler, and Fishwick 1998) are forming a bridge using DEVS as an intermediate formal specification language as the target.

There have been several attempts to unify modeling. One recent attempt is the creation of the Unified Modeling Language (UML) (Muller 1997; Lee 1997; Larman 1998). While UML may not unify all types of models ever required of the modeler, it does at least create a structured object-oriented architecture for certain types of phenomena (generally, software processes). VHDL (Bhasker 1995) serves to unify modeling for the VLSI community. The most common approach in modeling is to choose one or several model types and then to continue to use these types. As an inertial approach to modeling, this familiarity with a modeling language can cause problems when it comes to integrate with what others are doing. Often, the author of a particular modeling approach will advocate a singular model type over all other types. If the approach is found lacking, then it is augmented over time for completeness. Petri nets (Peterson 1981) are a good example of this strategy, with many variations over the original, surviving and prospering. There is no getting around the situation in which we currently find ourselves: there will always be many ways to define dynamic models and so we must work to describe the "glue" that allows models of varying types to be used together in a flexible information architecture. The architecture must be capable of the invention of existing and new model types, while still supporting a unified structure. While it is tempting to imagine that a unified modeling approach exists, this remains a dream and does not correspond to the practice and history of modeling. Our philosophy has been to formalize the *interfaces* and *glue* without getting into an argument of whether model X is better than model Y. It is most likely that model X is good for some tasks, and model Y for others. Sometimes, model X can be translated into model Y, which can make them complementary. One could argue that since one model can be proved to be equivalent to another under certain translations that one model or the other is superfluous. But, this is not the case—each model has its own character and delivers its own peculiar benefits for the modeler.

Objects know nothing of their surroundings except that they can receive input on *ports* and can post output to *ports*. The ports of an object serve as its formal interface to the outside world. We define three types of port: input, output and input/output. Input and output represent directional data flow, whereas input/output are appropriate for bidirectional energy flow as for bond graphs (Rosenberg and Karnopp 1983).

3 SEMIOTICS

With regard to representation of objects, we understand the need to create a digital object for each physical object in our scenario, but what about all of the models inside each object? Are they to be represented as data, methods or objects? We represent a model as an object, whether that model is geometric, dynamic or otherwise. Surfacing the model as an object is based on semiotic principles (Noth 1990) where one clearly separates form from interpretation,

Figure 2: Painting by Rene Magritte.

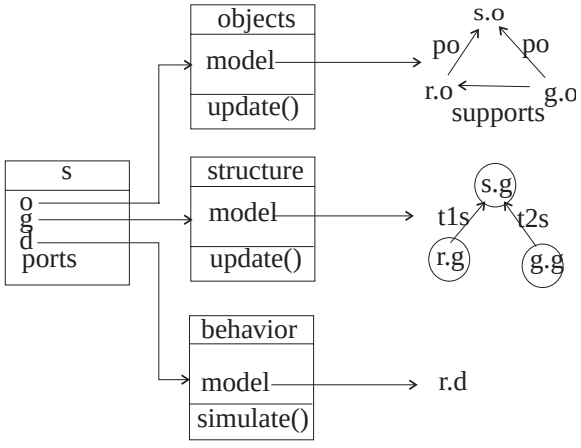

Figure 3: Scenario s Structure.

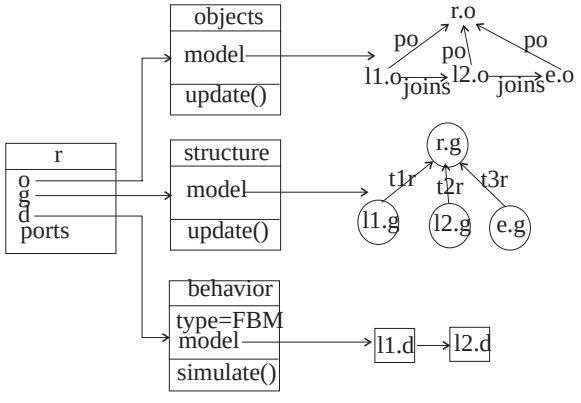

Figure 4: Robot r Structure.

or symbol from meaning. Figure 2 displays a painting, which captures the essence of semiotics. In the same sense as Magritte's painting "is not a pipe," likewise our model representations are more than formal model descriptions. They are physical objects containing model descriptions.

If the robot arm r moves through space according to forces applied, we may partition the space and call each partition a state of the robot. Then, we create a finite state machine (FSM) as a dynamic model $r.d$ inside of r. If this FSM is created from circles, arcs, and labels, we surface their importance as physical objects on an arbitrary physical medium (paper, pixels). The FSM is a model and also a physical object contained within the medium. Likewise, the diameter of the circles and the geometry of the arcs are an intrinsic part of the model, and not just of the Graphical User Interface (GUI). Therefore, the GUI and the model are one and the same. Not only are the models objects, but each model component is also an object—a dynamic model object.

Each model object contains an attribute called *model* and a set of methods applicable to the model. For example, if we wish to simulate a model, we should have a $simulate()$ method inside of the model object. The simulation applies to the model and not directly to the physical object. However, through aggregation, the physical object contains the model objects, and so, also contains the object's methods. Given strict encapsulation, we store methods such as $solve()$, $simulate()$, and $optimize()$ in the model objects.

Figures 3, 4 and 5 represent the architecture of the physical s, r and $l1$ objects. We begin with the scenario s in Figure 3. Like other physical objects, it is composed of a set of attributes, three of which are models. The three models are o for Object, g for Geometry and d for Dynamic. The value of $s.o$, for example, is an object with name $objects$ which is displayed to the right of s with a pointer from $s.o$. The model is then the attribute value of $objects.model$.

4 MODELS

A physical object is equivalent to a set of models plus attributes, where each model provides a different perspective on the nature of the object. To the extent that attributes of a physical object can be considered degenerate data or static models, a physical object is identical to the model set. There are as many types of models as necessary to accurately represent a digital object. Our models are encapsulated within objects, and they all have two key properties of recursion and multimodeling. The recursive property of all models states that a model is defined as a structure of models, and the multimodeling property (defined further in Sec. 6) states that each model component may be replaced by components of any model type. This methodology was introduced by Fishwick (Fishwick 1991) subsequently renamed "multimodeling," a

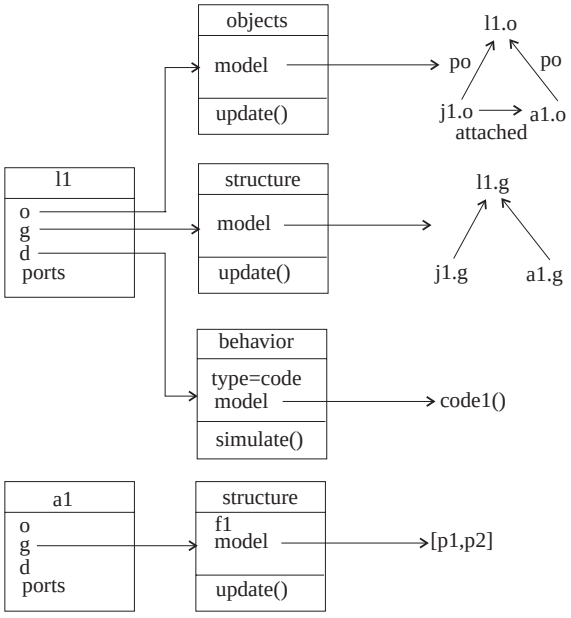

Figure 5: Link $l1$ and Arm $a1$ Structures.

term coined by Oren (Oren 1991), and then combined with Zeigler's DEVS formalism (Fishwick and Zeigler 1992). Recursiveness gives us hierarchy and multimodeling gives us heterogeneity. The following are model types that are frequently used in computer simulation, but the list is not exhaustive.

4.1 Object

An object model is a graph of objects with object identifiers as the graph nodes and relational arcs. The relations represent how objects relate to one another. For example, g supports the robot r and both r and g are part of (i.e., po) s. This model is identical in structure to the *object model* in UML. Method $object.update()$ operates upon $object.model$ as to be semantically driven. The two po relations may indicate an aggregate relation from attributes or methods in r and g to an attribute or method in s. The semantics can take any form that we like, and a relationship that is as broad as aggregation can be semantically equivalent to operations such as concatenation, summation or integration.

4.2 Geometry

A geometry model is a tree of geometry models, and so is recursively defined as for the object and dynamic models). The sort of tree is referred to as a hierarchical structure network (Foley, van Dam, Feiner, and Hughes 1990) in computer graphics. The labels $t1s$ and $t2s$ represent matrix transformation operations that must occur on the frames associated with the child geometry models so that the individual models are correctly positioned with the parent frame. The semantics $structure.update()$ is executed to ensure that these geometric transformations occur correctly. There other other types of models that can be used in geometry: voxels, particle systems, space-based partition trees and constructive solid geometry (Samet 1990b; Samet 1990a).

The termination of the recursion for geometry models is in the form of a data structure. In Figure 5, arm $a1.g$ has no sub-models, and is defined by a pair of points in the plane (defining the line segment $a1$ modeling the shape of the arm).

4.3 Dynamic

A dynamic model is a structure of dynamic models. Several types of dynamic models exist (Fishwick 1995) and are presented in a taxonomy similar to the one used for programming languages. Other taxonomies exist for formal model specification, such as those for DEVS (Zeigler 1990) and for the more general model development process (Overstreet and Nance 1985) In Figure 4, the behavior of r is defined by a functional block model (FBM) $behavior.model$. The termination of the recursion for dynamic models is in the form of a code method. In Figure 5, arm $a1.d$ has no sub-models, and is defined by a code method (function) then when executed, takes the output from $l1.d$ (in $r.d$, Figure 4) and feeds it to the input of $l2.d$ in the same FBM.

4.4 Conceptual

Classes are not central to the specification of objects, although they are useful for two reasons: 1) to permit the creation of *concepts*, and 2) to enable a useful mechanism for creating new objects from the class "template." If it were only for #2, we would not require the notion of a class since it is easy to create objects through natural (evolution) or artificial (manufacturing) means. One can clone an object or create an object by creating a physical object generator.

If we have a need to generate concepts, we can do so by creating a conceptual model on a physical medium. A conceptual model is similar to the class model in the classic object-oriented literature. In what physical object should a conceptual model be housed? One can create a medium such as paper or a blackboard and then place the conceptual model on it with all of its concept/class components. If we denote concepts using uppercase, R for robots and L for links, then we can create trees or graphs not unlike what we have done for object models. We let a concept be physically contained within that portion of the physical medium where it is specified. For paper, concepts

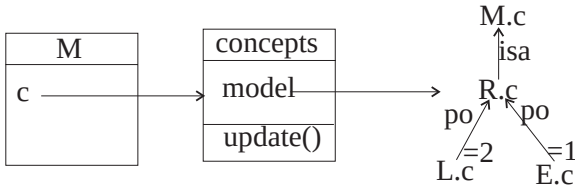

Figure 6: Mechanism M Structure.

are encapsulated within the sections of paper where they are drawn. While it may seem odd to characterize concepts as objects, this architectural necessity places conceptual models on an even keel with all other model types (i.e., all models are encapsulated within physical objects). The main difference between conceptual models and the others, is the conceptual model is referential—it refers to a set of concepts about something entirely different than the medium on which it is represented. The other models fit inside—and describe—their physical containers. Consider Figure 6. We define a *Mechanism* M concept and a concept for each object that we have created in Figure 3. A robot concept R isa kind of mechanism, and a robot has two links and one end effector. We can use these concepts to create specific objects, instead of manually specifying them as we did in the earlier figures. Therefore, $Rr1$ creates robot object $r1$ from concept R. This syntax is similar to that found in C++ and Java, with the concept being called a class, a a constructor used to generate new objects from a pre-specified class. $concepts.update()$ is used to operate on the model, and this is where one would insert semantics to deal with familiar operations such as specialization and inheritance.

4.5 From Code to Model

To increase the level of reusability, one can create models from logic that may appear in the model methods. Consider dynamic models with their $simulate()$ methods, operating on the models. It is also possible, through physical metaphor, to create a model of the simulation process itself, thereby extracting some of the semantics of $simulate()$ and surfacing these semantics in model form. The procedure of surfacing models in an architecture such as OOPM is a procedure where we are incrementally eliminating code structure, while creating new model structure to compensate.

5 ENCAPSULATION

A key aspect of digital objects is that information about an object should be encapsulated within that object. If link $l1$ weighs 10kg then there should be a "weight" attribute inside the $l1$ object. If $l1$ has a behavior represented by a dynamic model, then all components of that dynamic model must be encapsulated within $l1$. This prompts the question of how to "get information" from other objects into $l1$. Information within $l1$ must be 1) declared locally within $l1$, 2) enter $l1$ through its ports, or 3) be copied into $l1$ via a static relation between $l1$ and another object (i.e., as found in object models). This relational approach works similarly for classes.

Encapsulation is a relation that can be made explicit one of two ways within a model. Consider any object such as $structure$ in Figure 3. Both $model$ and $update()$ are encapsulated within $structure$. This form of encapsulation is represented in the rectangular box instead of explicitly in tree form as for the value of $model$. However, the concepts are similar. $s.g$ encapsulates both $r.g$ and $g.g$. The key difference is that when a labeled tree is used to demonstrate encapsulation, the tree arcs have semantics which dictate an operation to occur between parent and child. In the rectangular box encapsulation, there is no operation; the box is a *placeholder* for the internal structure.

Two frequently used encapsulation relations are composition and containment. They both represent one object encapsulating other objects, but the meaning is different and the semantics can be different. The robot r is physically composed of $l1$, $l2$ and e whereas a wooden crate might contain r but would not be composed of r. Composition tends to be represented with "value attributes," which are local to the object, whereas contained objects have independent lifetimes from their parent, and are referentially maintained in the parent. Large, flat tree structures can be created from the encapsulated structures, through tree traversal. This has the benefit of showing all relations at once, but the local, encapsulation knowledge is lost in the process.

6 COUPLING

Models are connected from model components, whether in a graph or set of equations. There are four types of components from the object-oriented perspective: 1) an attribute name/reference, 2) an attribute value/dereference, 3) a method name/reference, and 4) a method value/dereference. For example, an FSM is a network of attribute values with the attribute being the state. An FBM is a network of method names. A System Dynamics model is a network of attribute names (levels,rates) and method names (source,sink).

The primary concern with coupling is to ensure that one component fits into another and that their data types match. The "fitting" is made clear through the semantics for a model type. The $simulate()$ within an FSM object specifies how model execution is to occur, whereas the $simulate()$ inside of an FBM is

particular to the FBM type, and so the semantics are different. On the same level, a model's coupling is dictated by rules that define the semantics for that model. Between levels (inter-level coupling), one can perform homogeneous or heterogeneous decomposition. In homogeneous decomposition/refinement, the component of a model of type X is refined into another model of type X. In heterogeneous coupling (Fishwick, 1991) there isn't the limitation of matching model types, so we can refine the state of an FSM into a different target model type.

To ensure that coupling is accurate in the heterogeneous case, we first consider that models are composed of the four previously defined types of components. For each component type, we define the inter-level coupling as functional coupling. Each model component is a dynamic model object. In each dynamic model, there is an attribute *model*. This attribute points to either 1) another dynamic model object, or 2) a code method. The simplest way to ensure proper coupling between different model types is to create a uniform structure for models and their components. For example, if every model and model component is a method, then we are assured proper coupling through recursive substitution of model for model component. Unfortunately, the method is not the best choice since models are objects, and require attribute information. Therefore, we achieve coupling by recognizing that to execute a model is to begin scheduling of an *event chain* with model components comprising the events. Ultimately, we terminate each dynamic model with a scheduled code method. All dynamic models are event scheduled via the $simulate()$ method.

7 SUMMARY

We have specified some steps for a new information architecture where digital objects, and their encapsulated models, are defined. The architecture is similar to the efforts of UML, especially with the focus on visual structures. However, the key differences are that our architecture is focused on physical object and model representation (and not software design), multimodeling (for heterogeneous coupling), and the "open architecture" ability to choose new model types at any time. With regard to existing standards for physical objects, one wonders where OOPM fits. For standards such as VHDL, we envision a situation where one could use the VHDL model type as a dynamic model type and then acquire methods that operate on VHDL models. This requires a slight shift in attention for software tool builders so that they build methods that are meant to be encapsulated (through inheritance, aggregation or direct insertion). The architecture presented is general enough to accommodate VHDL model types or any other model type as long as the code (methods) and models are clearly separated, and yet encapsulated within their respective physical objects. We end with some stated rules of this architecture:

- An object contains attributes and methods.
- A digital object has a one-to-one map with a physical object, and contains models for many, if not all, of its attributes.
- All models and model components are objects, and these are *equivalent* to their graphical or textual representations.
- Each model object contains an attribute "model" pointing to the model structure, and a set of methods applicable for that model.
- Multimodeling is the principle where one model component object may be replaced by component objects of any model type.

ACKNOWLEDGMENTS

I would like to thank the following agencies that have contributed towards our study of modeling and simulation: (1) Jet Propulsion Laboratory under contract 961427; (2) GRCI, Inc. and Rome Laboratory, Griffiss Air Force Base, New York under GRCI contract 1812-96-20; and (3) Department of the Interior under grant 14-45-0009-1544-154.

REFERENCES

Barros, F. J., B. P. Zeigler, and P. A. Fishwick. 1998. Multimodels and Dynamic Structure Models: An Integration of DSDE/DEVS and OOPM. In *1998 Winter Simulation Conference*, Washington, DC. To be published.

Bhasker, J. 1995. *A VHDL Primer: Revised Edition*. Prentice Hall.

Cubert, R. M., and P. A. Fishwick. 1997. MOOSE: An Object-Oriented Multimodeling and Simulation Application Framework. *Simulation*. To be published.

Fishwick, P. A. 1991. Heterogeneous Decomposition and Coupling for Combined Modeling. In *1991 Winter Simulation Conference*, Phoenix, AZ, 1199 – 1208.

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

Fishwick, P. A. 1996. Extending object oriented design for physical modeling. *University of Florida/CISE Technical Report*. http://www.cise.ufl.edu/∼fishwick/tr/tr96-026.html.

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

Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes. 1990. *Computer Graphics: Principles and Practice*. Addison-Wesley. Second Edition.

Larman, C. 1998. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall.

Lee, R. C. 1997. *UML and C++: A Practical Guide to Object-Oriented Development*. Prentice Hall.

Muller, P.-A. 1997. *Instant UML*. Olton, Birmingham, England: Wrox Press, Ltd.

Noth, W. 1990. *Handbook of Semiotics*. Indiana University Press.

Oren, T. I. 1991. Dynamic Templates and Semantic Rules for Simulation Advisors and Certifiers. In P. Fishwick and R. Modjeski (Eds.), *Knowledge Based Simulation: Methodology and Application*, 53 – 76. Springer Verlag.

Overstreet, C. M., and R. E. Nance. 1985, February). A Specification Language to Assist in Analysis of Discrete Event Simulation Models*Communications of the ACM* 28(2), 190 – 201.

Peterson, J. L. 1981. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Rosenberg, R. C., and D. C. Karnopp. 1983. *Introduction to Physical System Dynamics*. McGraw-Hill.

Samet, H. 1990a. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley.

Samet, H. 1990b. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.

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

AUTHOR BIOGRAPHY

PAUL A. FISHWICK is Professor of Computer and Information Science and Engineering at the University of Florida, and will be serving as General Chair of the 2000 Winter Simulation Conference. 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 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 Fellow of the Society for Computer Simulation, and a Senior Member of the IEEE. Dr. Fishwick founded the `comp.simulation` Internet news group (Simulation Digest) in 1987, which now serves over 15,000 subscribers. He has chaired workshops and conferences in the area of computer simulation, and will serve as General Chair of the 2000 Winter Simulation Conference. 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*. Dr. Fishwick's WWW home page is `http://www.cise. ufl.edu/~fishwick` and his E-mail address is `fishw ick@ cise.ufl.edu`.