Toward a Convergence of Systems and Software Engineering

Paul A. Fishwick
Computer & Information Science and Engineering Department
University of Florida

Abstract

Recent trends in software engineering, especially within the object-oriented community, reflect a clear trend toward systems engineering methods. Object oriented designs, meant for programming design, often take on the distinct appearance of system *models* of physical networks and devices. It is not immediately apparent how the areas of "modeling" and "programming" relate to one another, and why the convergence is taking place. To explore the convergence in depth, we discuss common concepts between models and programs, and discuss future trends in computer science which are forging a steady convergence between models and programs. The convergence is spawned by increased emphasis on object oriented design, distributed systems, complex systems, new forms of analog computation, and abstraction methodology. We close with a discussion of MOOSE, which provides a comprehensive modeling environment for both programmers and modelers.

1 Introduction

Scientists and engineers use mathematical models frequently to avoid having to revisit the drawing board whenever new designs are planned or a physical device is to be constructed. Models are patterns of behavior and structure in physical systems. By creating a model of a system, a scientist is able economically record and abstract the behavior and geometry of a physical system at various abstraction levels. While the use of modeling has been utilized in software [42, 9, 13] and systems engineering [49], modeling requires a stronger base in software engineering. Our purpose is to relate the use of *model* to *program* and to justify the need for increased attention to modeling in software systems. We achieve this goal by discussing the relationship between models and programs. The effort of tying models and programs closer together has been studied by a number of researchers. In the simulation community, Nance, Balci and Overstreet [35, 5, 39] have focused their modeling studies in the bridge area of software engineering and simulation modeling. Oren [38], Zeigler [57, 58], Cellier [12] and Fishwick [16] espouse modeling methods and formalisms which can be cross-referenced in either simulation or computer science-based systems. Let's consider a simple dynamic model where Fig. 1 illustrates a manufacturing floor where parts are provided to a spiral accumulator and then fed through two workstation cells which machine parts by performing lathe and drill operations. The raw stock arrives from the left via a central conveyor. This type of application involves discrete parts flowing through a network of resources which proceed when constraints are met. This suggests the use of a Petri net to model the system in Fig. 2.

The first figure is a conceptual model and the second is a dynamic model. Fig. 1 can be considered a model which is non-executable, but descriptive of the overall topology of the physical setup. The second model is executable and employs one of many possible dynamic

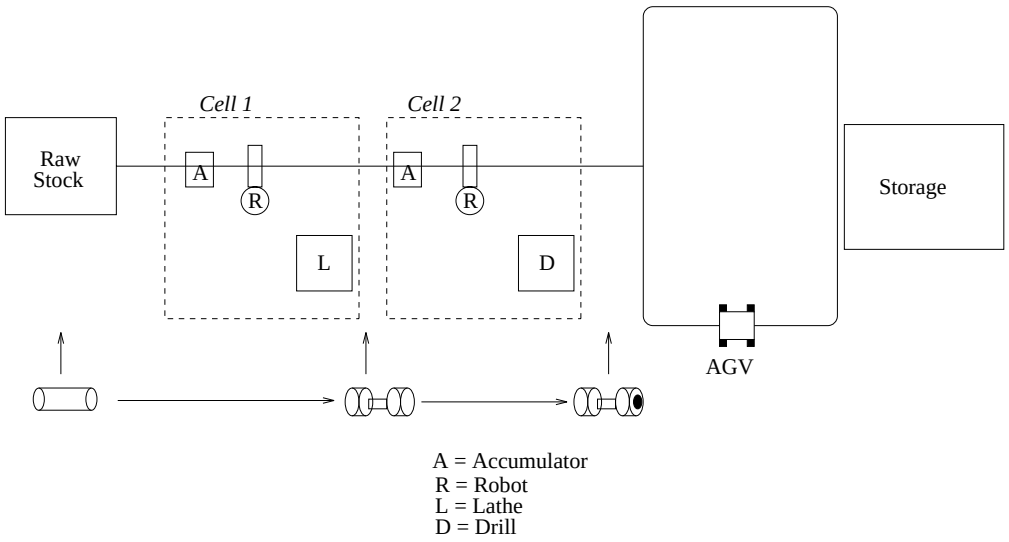


Figure 1: Manufacturing line with two robots and two machines.

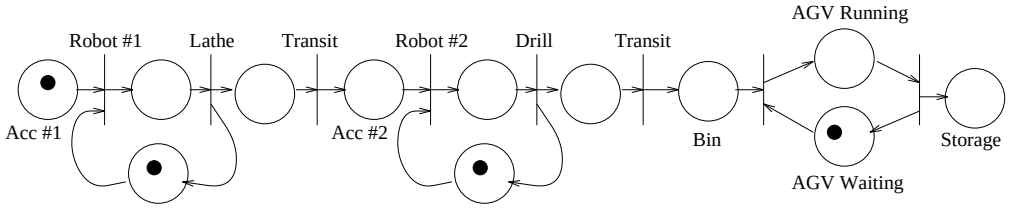


Figure 2: Petri net for manufacturing line.

model types. We can see the potential use of these models in the construction of large software and system parts. For instance, if we are constructing software to manage and control the physical manufacturing devices, each of which will also have embedded digital circuits, there is a need to abstract the system into a model to better understand the relation between flowing parts and the reactive mechanical pieces.

Models can be constructed as visual or textual artifacts. Sample model types include finite state and event automata, Petri nets, logic/production-rule systems, functional block models, equation sets and spatial models [19]. The model serves as a mental device for reasoning about the system without necessarily performing all experiments on the full system. Moreover, the model is generally cheaper to construct and maintain and serves as a pattern used for archiving and disseminating system knowledge.

Given the importance of modeling, we find similar approaches in use throughout computer science—especially within software engineering. There is a convergence within software and system engineering due to the strong ties between programming and modeling. By exploring the nature of this convergence, we will strengthen the idea that system modeling is a useful enterprise for software engineers and that software engineering trends—particularly in object oriented design—are beneficial to system engineers. We limit our discussion of models to *dynamic* models even though the use of *geometric* models[1] plays an equally critical role in systems design. The design of models involving time (i.e., dynamic models) forms the basis of the study of *computer simulation*. It is in the field of computer simulation that many of the converging properties of systems and software engineering are found.

Our presentation will be as follows. First, we will discuss the positive impact of object oriented design on program development and discuss limitations of object oriented design and where improvements can be made. Subsequently, we explore the interrelationship of modeling and programming. We attempt to answer the question of "How is a model different from a program?" We proceed to a new model taxonomy for systems design and engineering using several principles of computer programming languages. MOOSE (Multimodeling Object Oriented Simulation Environment) reflects our prototype implementation of the available modeling types. We close the article with recommendations on how computer scientists can play a more significant role in the design and representation of models.

2 Object Oriented Design

In object oriented (OO) design, there is a natural focus on the classification of objects. Objects are created as instances of classes, classes are joined together in a tree-like fashion to form class hierarchies, and classes are subsequently endowed with attributes and methods. The implementation of Simula[7] (an early simulation language which is still thriving, especially in Europe) is considered the first language which fully exploited the benefits of OO programming. Benefits, often touted in OO circles, include code reuse through inheritance and polymorphism; however, the most important aspect of OO design and implementation is that OO design embodies a fundamental principle: *think in terms of the real world application first: physical objects, attributes and methods.* That is, if we are building a system

[1]Geometric modeling approaches find coverage in areas such as computer graphics, computer aided design and computational geometry.

involving air traffic control, then we should build our program using physical objects: aircraft, control tower, computer consoles. It could be argued that, while a focus on real world objects is perfect for simulation applications, software applications present a different story. However, given this issue, consider a sample of recent literature on software design using OO principles [8, 10, 47, 51, 52]. This OO emphasis is on models of *physical systems* such as ATM machines and air conditioning control, and elicits the following question: "Are software engineers doing system engineering?" The OO designs described are of physical systems. To more fully appreciate why software engineering is orienting itself more toward modeling, we need to define some terms:

- *Software Engineering*: the process of creating software and specifications from concept to executable software (in the form of a program), and continuing to analysis and maintenance of the software.
- *Systems Engineering*: the process of creating models and specifications for physical systems.
- *Dynamic Model*: a structure which serves to abstract the behavior of a physical system.
- *Geometric Model*: a structure which serves to abstract the geometric configuration of a physical system.
- *Program*: an implemented algorithm which executes on a computer of some sort (analog or digital).
- *Computer*: a physical device capable of taking a program and executing it to produce output. In our terminology, "computer" is a very general concept and could refer to the Babbage differential analyzer, a computer created from tinkertoys [14] or a modern digital computer.

Sometimes, the terms "model" and "program" are used interchangeably. In a formal sense, models and programs can be interpreted as equivalent, but the terms are quite overloaded and so we will need to examine them more carefully. To better explore the reasons why there is a trend in integrating modeling, software and systems engineering, we need to dig a little deeper in the following sections.

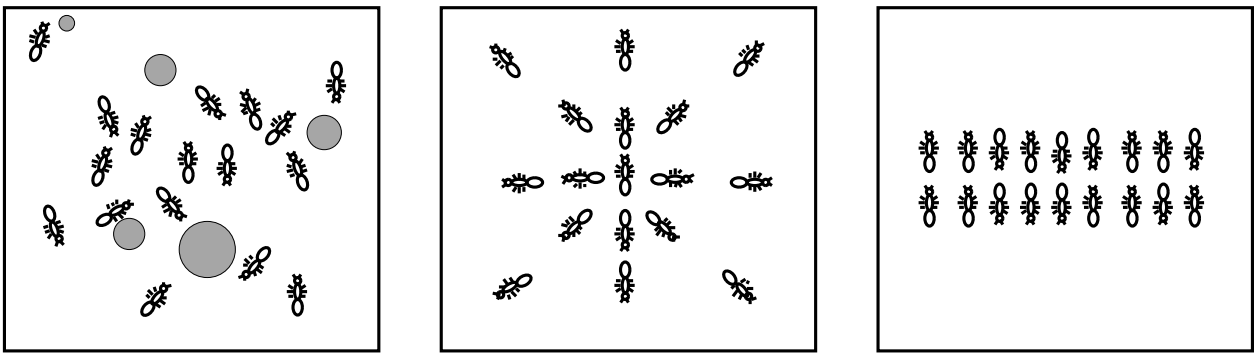
3 Model + Computability = Program: From Models to Programs

Computer programs are models which exhibit a high level of computability and control. This statement requires some discussion and justification since the reader may ask "What physical process, if anything, is the average computer program modeling?" Programs that execute on computers model the physical attributes and devices of which computer is constructed. For instance, when a digital electronic computer is used to execute a program which sorts database records using an index field, this program causes transistors to switch and media to become magnetized or scanned. This complicated sequence of *physical* transformations represents the system hardware being modeled. The main reason why we might not think of the sorting program as a model of electromagnetic behavior is that we have so much *control*

over the system that we have essentially forgotten the physical substrate completely. The physical processes occurring while the program is executing have been "abstracted away." For all we care, the underlying machine could be made up microscopic animals. As long as we get consistent and timely output from the program, we will be satisfied that our programming time is wisely spent.

We begin to think of programs as models when the level of control is relaxed and there are uncertain elements injected into the physical system. Consider three scenarios in Fig. 3. These scenarios are based on "artificial ants" which are provided to illustrate the previous point about models and programs. Fig. 3(a) displays the current state of a small ant colony. The ants may be foraging for food among obstacles or using a pheromone trail laid down by ants who have located the food. We can build a model for this physical system using several methods. One approach would be to assign individual behaviors to each ant, allowing the ant to turn and move according to sampled random distributions. Ants move around or over obstacles they find in their way. This physical system might not make a good computer on which to run a program since (1) the dynamics of the ant colony does not exhibit Turing equivalence [31], but more generally, (2) we are not in sufficient *control* of the ants. When we say "control," we are referring to a combination of the system property *controllability* [27] and the concept *effective procedure* [2] which embody what we normally think of with a modern "program."

Suppose now, that we change the ants so that they always move in a two-dimensional, random fashion (random walk) and that we now have the ability to move or stop an an ant at our discretion. Fig. 3(b) displays this situation after we originally cluster the ants in a solid circle and then let them walk for a given time. We have removed the obstacles for a more controllable situation and to achieve some level of regularity. The ants now perform a useful calculation called *diffusion*. Using the ants in this way, we may now model heat flow on a two-dimensional metallic plate with an initial circular boundary heat source. This is a tremendous capability, computationally, since many thermodynamic calculations may take place with the ability to model a wide variety of physical phenomena, including diffusion-limited aggregation and reaction-diffusion properties. We are now in more control of the ants than before with the scenario in Fig. 3(a), and we have created another model this time, not of the ant behavior alone, but of the behavior of heat flow. At this point, we could certainly claim that we are "programming" instead of just "modeling" because of the regularity associated with the ant behavior. Fig. 3(c) displays a situation where we are in even further control of the ant motion. Ants are in one of two possible states: up or down. Ants now occupy grid points and can be moved from one point to another arbitrarily through an electrical signal issued to the ant via grid voltages. This final ant scenario provides a cellular automaton (CA)-type of capability. The ants are in one of two states: up (1) or down (0), so we achieve a binary encoding. A partially bounded 2D CA can be shown to have Turing equivalence since it can be divided into a finite number of 1D CAs [53], thereby making the ants as computationally powerful as the modern general purpose computer. With this new level of control and computability, we can program the ants to behave as we prefer. The movement from Fig. 3(a) to Fig. 3(c) can be seen in either direction: gradually gaining ant control or losing control. Ashby [3] refers to the loss or gain in regulation (control) of a system as a change in *requisite variety*. Although this discussion provides us with one way of better relating models with programs, we should note that programming has branched out



(a) Ants in a natural environment (no controls)

(b) Ants performing random walk (distributed control)

(c) Ant computer with Turing equivalence (global control)

Figure 3: Ant models and programs

to the point where *programs* created using adaptive complex system models, such as genetic programs and neural network programs, have blurred the distinction further between model and program. With a more de-centralized type of control, we could program ants to achieve key "programmed" results through mutation, gene selection and fitness functions. However, we still would need to be able to exert control on individual ant behaviors even if the collective orchestration of ant behavior is not performed. Analog computation enjoys a much richer history than digital computing and also serves to make the idea of model and program one and the same. Electronic analog computers [4] were constructed to solve differential equations much like Babbage's engine was built to solve polynomials through differencing. The programs for these computers are models in every sense since the program was built to model physical phenomena. Recent work in analog computing falls into the category of *analog VLSI computation* [33] where programs are constructed to manipulate the parameters for neural networks, which perform a useful function, such as emulating biological sensors, when executed. Many other analog computations are possible [14] including those associated with DNA-based computing [1].

4 Program + Metaphor = Model: From Programs to Models

Now, that we can claim that any program is a model of the underlying physical structure of the computer, let's consider the conversion of algorithms or programs to models through the liberal use of *metaphor*. Programs which code models of systems do not really require a metaphor because they already have a direct connection to the real world. For instance, if I write an event scheduling program which codes the actions of a program to simulate a line forming at a grocery store cashier counter, then my program is a model. That is, it is something that requires interpretation or compilation and is subsequently executed. However, if we have a program which sorts N numbers, how can this be considered a model[2]? Many sorting and searching operations have analogs in the physical world. For instance, if our sorting program does *MergeSort* then, our program can be seen as a model for customers

[2]Ignoring, for the moment, the general method pointed out in the previous section, where we were modeling the physical computer elements.

who are in two queues originally, but are then told they must form a single line due to a malfunctioning cashier's machine. Plates stored in cafeteria spring-loaded compartments are metaphorical equivalents of stack-based program and data structures. Numerous other instances exist where—if we take a program and apply metaphor—we change the meaning of the program to be one involving real world semantics.

5 Reasons for the Model ⇔ Program Convergence

Programs and models were not always distant cousins; they grew up together. Prior to the advent of digital computers, most computers were modulated (controlled) by programs which we think of as models today. The electronic analog computer contained patch panel programs which we classify as *constraint models* in the form of differential equations. Programs were constructed physically by patching together processing elements via coated wires on perforated boards. Areas such as automata theory and cybernetics [3] paid special attention to the relations of model to program. Today, a program is generally considered to be a tool with a certain degree of freedom. Chomsky's hierarchy of language complexity—which can be shown equivalent to levels of computational complexity—provides us with the power we normally associate with programs. Generally, a modern program is considered to have the power and expression of a Turing machine.

We have discussed how models and programs are very much like each other, sometimes after the application of attributes such as control, computability and metaphor. But why should the modeling⇔programming connection arise at this particular point in the history of computer science? One reason is that we are becoming more liberal about how we define "program." Also, the concepts of programs and hardware are merging. Some of the areas accelerating this convergence of "model as program" are:

- *Modern Analog Computation:* Computation started originally on analog computers and then later proceeded to digital computers. What was wrong with the programs on the analog computers? One problem was (and still is) that fairly large scale analog components have varying operating regions which affect the computation, say, of a differential equation, even though the actual computation may be blindingly fast. The old patch panel method of programming has given way to better user interfaces with computers which have higher degrees of reliability. The analog VLSI approaches now being used [33] are more reliable than the operational amplifiers used on the original analog systems.

- *Distributed Computation:* With a mainframe system, a program runs on a single central processing unit (CPU). Now that systems are becoming increasingly distributed, we are finding that computing elements are literally in every physical system, including car engines, traffic lights, keyless entry systems, and even doors. Negroponte, in his recent book [36], expounds upon this trend from a computer-human interface perspective. Distributed hardware implies that software must be distributed as well. As the single program (once run on a single CPU) is gradually divided into its disparate elements, each associated with a physical device, the programs take on the appearance of physical models of the distributed system.

- *Emergent Programs:* The idea of evolving a program to solve a particular problem is quite different from the usual software engineering approach of doing requirements analysis and then writing a specification. Programs can be looked at like models with embedded control so that if we want to build a program to solve problem X, we use an optimization approach (in conjunction with genetic algorithms [22, 29] or neural networks) to "evolve" the program. The program becomes a model of biologically evolutionary behavior. The work in *artificial life* [30] capitalizes on these modeling techniques.

6 Computer Scientists as Model Builders

The overlaps between modeling and programming suggest that computer scientists take a pro-active role in learning and applying the technique of modeling. An OO taxonomy containing only one kind of dynamic model (finite state machine) is limited, and so this will require expansion to include several types of models (see Sec. 8), many of which include both discrete as well as continuous system properties. Here is a list of suggested key focus areas to facilitate a concentration in model building:

1. *Software/Systems Engineering:* The convergence of models with programs means that there is a corresponding convergence between software and systems engineering principles. Hardware and software are integral system components and both need to be modeled under a unified theme of *system modeling*. The common formal base of all systems is systems theory [6, 40], which leads to systems science [21] and engineering [49].

2. *Object-Oriented Design:* The OO philosophy promotes a clean interface between real world objects and computer programs, and this is one of the key reasons why the OO paradigm is successful. One thing we need to stress: perform an object oriented design *before* thinking of implementation issues such as what particular language features are to be exploited. Sometimes, the teaching of C++, for instance, is thought of as equivalent to object oriented design. Instead, we need to teach design first, and program later. One of the best ways to do this is to encourage the liberal use of modeling. By constructing models before looking at code properties, we develop the right skills to model.

3. *Curriculum Development:* With an increasing use of computers by scientists and engineers, other disciplines will look toward computer scientists for help in modeling as well as assessing algorithmic needs such as reducing computational complexity and achieving parallelism. The computer science curricula need updating to include a heavy dose of modeling. Computer science curricula tend to stress discrete systems over continuous ones to the point of ignoring real world phenomena. Courses should put more stress on modeling, computer simulation and systems theory (a unified view of systems). The use of interdisciplinary approaches in solving systems problems should be highly encouraged. A healthy trend toward this interdisciplinary view is taken in *computational science and engineering* (CSE) [45]. Modeling should be introduced in core computer science classes.

7 Abstraction

Models come in all flavors, and most model types are adept at describing system dynamics at some specific abstraction level. Differential equations would normally be used to model low level physical phenomena, and would probably be the lowest level of abstraction for most systems. Finite state automata, on the other hand, would be used to represent higher levels. Abstraction is used in programs as well with regard to data and program abstraction. Moreover, the topic of code translation can also be seen in light of abstraction levels. For programs, we might build a compiler which translates C source into assembly language instructions. For models, we may have a Petri net which translates into equation sets. The ant system in Fig. 3 serves as a springboard for example translations involving programs and models: (1) physical ants modeling diffusion; (2) a digital computer interpreting a computer program which simulates virtual ants where the ants model diffusion; or (3) A C program translated into assembly code which is executed on an ant computer. There are an endless number of transitive couplings in which models and programs are interweaved to form chains of translation or abstraction. Abstract models usually take less time to execute than more detailed models and they present a different kind of information. All physical phenomena are models of themselves and all dynamics of those phenomena are simulations to the limit of the actual system. It is not practical to use an actual system to simulate itself since we do not reduce *cost*, whether cost is measured in monetary units or time. However, seeing modeling in this light permits us to see how abstraction operates over multiple scales. The parallels between modeling and programming abstractions leads to other questions such as one involving input and output. A program can be input to another program, which operates upon it and produces a program as output. This procedure is common to compilers and interpreters but what about in modeling? Normally, models associate *signals* with both input and output, but these signals are just attributes of physical objects (in the OO sense) which are being transported. Consider a machine which takes two pieces of plastic and a spring and then assembles these objects into an assembled output object. Each physical object has different dynamics. Thus, models can take models as input and produce models as output. Even when the dynamics of an object do not change, as when a customer object passes through a cashier resource, objects and their encapsulated models are accepted as input and sent along to the output.

8 Modeling

Programs and models have historical differences in that programs have tended to be non-visual in nature. There are counter-examples to this trend, such as the work in visual programming languages and recent OO program visualizations [23, 13], but overall the field of modeling has always focused heavily on visual formalisms. To a great extent, the topology in models correlate to the distinct geometric structures in natural and artificial systems: we construct icons where we see physical pieces in the real-world system. Just as the discipline of software engineering has emerged to address this question for software, in general, modelers also have a need to explore similar issues: *how do we engineer models*? While there are many modeling techniques for systems and simulation, we are often in a quandary as to which model technique to use, and under what conditions we should use it. Our approach

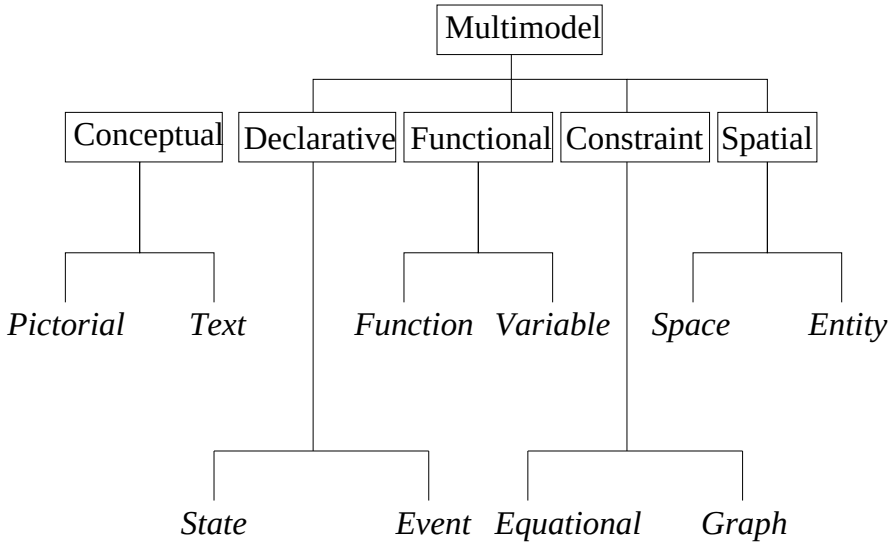


Figure 4: Taxonomy for modeling.

is depicted in Fig. 4. This taxonomy of modeling requires some justification before we describe the model types. The taxonomy was created by considering a large number of existing modeling methods in science and engineering. Through induction and clustering, we proceeded to form categories. Conceptual modeling has its roots in object oriented design, artificial intelligence and data base management systems. Declarative, functional and constraint modeling methods are very similar to formal programming language types which have the same names (i.e., declarative semantics, functional languages, constraint logic languages). The spatial modeling type was gleaned mostly from physics, and the multimodeling type was designed to include those hierarchical models found most often in digital circuit design and in *combined* discrete-continuous simulation models.

We will briefly discuss the model types in Fig. 4. A more complete written treatment is provided in [19]. There are five basic model types, and one complex model type which includes abstraction levels, each composed of one of the basic types. Conceptual models represent the first phase in any modeling endeavor. All static and dynamic knowledge about the physical system must be encoded in some form which allows specification of interaction without necessarily specifying the dynamics in quantitative terms. Semantic networks [56] present one way of encoding conceptual semantics; however, we have chosen object-oriented design networks [10, 47] which have more formal treatment. The ultimate conceptual model is one based on database technology, such as an object-oriented database, capturing all facets of the physical system.

Declarative models permit dynamics to be encoded as state-to-state or event-to-event transitions. The idea behind declarative modeling is to focus on the structure of state (or event) from one time period to the next, while de-emphasizing functions or constraints which define the transition. Models such as finite state automata [25], Markov models, event graphs [50] and temporal logic models [34] fall into the declarative category. Declarative models are state-based (FSAs), event-based (event graphs) or a hybrid (Petri nets [41]).

Functional models represent a directional flow of signal (discrete or continuous) among

transfer functions (boxes). When the system is seen as a set of boxes communicating with messages or signals, the functional paradigm takes hold. A "data flow model" is a functional model. The use of functional models is found in control engineering [37, 15] (with continuous signals) as well as queuing networks for computer system model design [32]. Some functional systems focus not so much on the functions, but more on the variables. Such models include signal flow graphs, compartmental models [26], and Systems Dynamics [46].

There are two types of constraint models: *equational* and *graph-based*. Constraint models are models where a balance (or constraint) is at the heart of the model design. In such a case, an equation is often the best characterization of the model since a directional approach such as functional modeling is insufficient. Equational systems include difference models, ordinary differential equations, and delay differential equations. Graphical models such as bond graphs [11, 28] and electrical network graphs [44] are also constraint based.

If a system is spatially decomposed as for cellular automata [55, 54, 24], Ising systems, PDE-based solutions or finite element models, then the system is being modeled using a spatial modeling technique. Spatial models are used to model systems in great detail, where individual pieces of physical phenomena are modeled by discretizing the geometry of the system. Spatial models are "entity-based" or "space-based." Entity-based spatial models focus on a fixed space where the entity dynamics are given whereas space-based focus on how the space changes by convolving a template over the space at each time step. PDEs are space-based where the template defines the integration method. L-Systems [43] are entity-based since the dynamics are based on how the organism grows over a fixed space.

Large scale models [48] are built from one or more abstraction levels, each level being designed using one of the aforementioned primitive model types. The lowest level of abstraction for a system will probably use a spatial model whereas the highest level may use a declarative finite state machine. Intermediate levels will often use functional and constraint techniques. Models which are composed of other models are termed *multimodels* [20, 16, 18]. By utilizing abstraction levels, we can switch levels during the simulation and use the abstraction most appropriate at that given time. This approach gives us multiple levels of explanation and is computationally more efficient than simulating the system at one level.

9 MOOSE Implementation

We are developing a Multimodeling Object Oriented Simulation Environment (MOOSE) to allow users to construct models interactively using many abstraction levels. There are three levels to MOOSE: (1) Graphical User Interface (GUI); (2) Blocks Language; and the (3) SimPack simulation toolkit. The GUI is developed using Tk/Tcl and serves to interface with the user via three windows: experiment,modeling,scenario. It is possible to construct multimodels interactively. These models allow a heterogeneous model construction capability. The Blocks language is used as an intermediate "assembly language" and SimPack [17] serves as the base simulation library toolkit which executes all Blocks models. SimPack currently has over 180 users worldwide and is documented in `http://www.cis.ufl.edu/~fishwick` under "SimPack Simulation Toolkit." Other simulation information on MOOSE is located at this URL as well. The current status of the MOOSE project is that all three major components are working for single-level functional block models. We are currently implementing the multimodeling capability.

10 Summary

Programming will be aided by model-based approaches, such as dynamic modeling, to help software engineers construct large systems. We have examined the converging fields of modeling and programming to illustrate the need for more attention to this growing area of intersection. Software and systems engineers will work closely together using the same modeling techniques, but perhaps at different abstraction levels. All levels are connected together to form a multi-level large-scale model (a multimodel). In this synergistic *systems view* of the world, models can be abstracted and refined to the limit where they are equivalent to programs. This convergence has been accelerated by several factors including distributed computation and real-time systems, object-oriented design methodology, complex system modeling, evolutionary computing and abstraction methodology. Since modeling—during the next century—will likely take up a significant percentage of all computer time, computer science curricula must respond to the challenge by instrumenting courses in modeling and systems study. As a stepping stone toward the objective of using a multimodeling construction tool, we have begun development on the MOOSE system.

Acknowledgments

I would like to acknowledge the following funding sources which have contributed towards our study of modeling and implementation of the MOOSE system: (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

[1] Leonard L. Adleman. Molecular Computation of Solutions to Combinatorial Problems. *Science*, 266:1021–1024, November 1994.

[2] Michael A. Arbib. *Brains, Machines and Mathematics.* Springer Verlag, second edition, 1987.

[3] W. Ross Ashby. *An Introduction to Cybernetics.* John Wiley and Sons, 1963.

[4] J. Robert Ashley. *Introduction to Analog Computation.* John Wiley and Sons, 1963.

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

[6] Ludwig von Bertalanffy. *General System Theory.* George Braziller, New York, 1968.

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

[8] Grady Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, 12(2):211 – 221, February 1986.

[9] Grady Booch. On the Concepts of Object-Oriented Design. In Peter A. Ng and Raymond T. Yeh, editors, *Modern Software Engineering*, chapter 6, pages 165 – 204. Van Nostrand Reinhold, 1990.

[10] Grady Booch. *Object Oriented Design*. Benjamin Cummings, 1991.

[11] Peter C. Breedveld. A Systematic Method to Derive Bond Graph Models. In *Second European Simulation Congress*, Antwerp, Belgium, 1986.

[12] Francois E. Cellier. *Continuous System Modeling*. Springer Verlag, 1991.

[13] Peter Coad, David North, and Mark Mayfield. *Object Models, Strategies, Patterns & Applications*. Yourdon Press Computing Series. Prentice Hall, 1995.

[14] A. K. Dewdney. *The Tinkertoy Computer and Other Machinations*. W. H. Freeman and Co., 1993.

[15] Richard C. Dorf. *Modern Control Systems*. Addison Wesley, 1986.

[16] Paul A. Fishwick. An Integrated Approach to System Modelling using a Synthesis of Artificial Intelligence, Software Engineering and Simulation Methodologies. *ACM Transactions on Modeling and Computer Simulation*, 1992. (submitted for review).

[17] Paul A. Fishwick. Simpack: Getting Started with Simulation Programming in C and C++. In *1992 Winter Simulation Conference*, pages 154–162, Arlington, VA, December 1992.

[18] Paul A. Fishwick. A Simulation Environment for Multimodeling. *Discrete Event Dynamic Systems: Theory and Applications*, 3:151–171, 1993.

[19] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, 1995.

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

[21] Robert L. Flood and Ewart R. Carson. *Dealing with Complexity: An Introduction to the Theory and Application of Systems Science*. Plenum Press, 1988.

[22] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1989.

[23] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514 – 530, May 1988.

[24] Brosl Hasslacher. Parallel Billiards and Monster Systems. In N. Metropolis and Gian-Carlo Rota, editors, *A New Era in Computation*, pages 53–65. MIT Press, 1992.

[25] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[26] John A. Jacquez. *Compartmental Analysis in Biology and Medicine.* University of Michigan Press, 2nd edition, 1985.

[27] R. E. Kalman, P. L. Falb, and M. A. Arbib. *Topics in Mathematical Systems Theory.* McGraw-Hill, New York, 1962.

[28] Dean C. Karnopp, Donald L. Margolis, and Ronald C. Rosenberg. *System Dynamics.* John Wiley and Sons, 1990.

[29] John Koza. *Genetic Programming.* MIT Press, 1992.

[30] Christopher Langton, editor. *Artificial Life.* Addison Wesley, 1987.

[31] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, 1981.

[32] M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools.* MIT Press, 1987.

[33] Carver Mead. *Analog VLSI and Neural Systems.* Addison Wesley, 1989.

[34] Ben Moszkowski. *Executing Temporal Logic Programs.* Cambridge Press, Cambridge, 1986".

[35] Richard E. Nance. Modeling and Programming: An Evolutionary Convergence, April 1988. Unpublished overheads requested from author.

[36] Nicholas Negroponte. *Being Digital.* Knopf, 1995.

[37] Katsuhiko Ogata. *Modern Control Engineering.* Prentice Hall, 1970.

[38] Tuncer I. Oren. Model-Based Activities: A Paradigm Shift. In T. I. Oren, B. P. Zeigler, and Elzas M. S., editors, *Simulation and Model-Based Methodologies: An Integrative View*, pages 3 – 40. Springer Verlag, 1984.

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

[40] Louis Padulo and Michael A. Arbib. *Systems Theory: A Unified State Space Approach to Continuous and Discrete Systems.* W. B. Saunders, Philadelphia, PA, 1974.

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

[42] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw Hill, 1992.

[43] Przemyslaw Prusinkiewicz and Aristid Lindenmeyer. *The Algorithmic Beauty of Plants.* Springer-Verlag, 1990.

[44] R. Raghuram. *Computer Simulation of Electronic Circuits.* John Wiley, 1989.

[45] John R. Rice. Computational Science and the Future of Computing Research. *IEEE Computational Science & Engineering*, pages 35–45, Winter 1995.

[46] Nancy Roberts, David Andersen, Ralph Deal, Michael Garet, and William Shaffer. *Introduction to Computer Simulation: A Systems Dynamics Approach.* Addison-Wesley, 1983.

[47] James Rumbaugh, Michael Blaha, William Premerlani, Eddy Frederick, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[48] Andrew P. Sage. *Methodology for Large Scale Systems.* McGraw-Hill, 1977.

[49] Andrew P. Sage. *Systems Engineering.* John Wiley and Sons, 1992.

[50] Lee W. Schruben. Simulation Modeling with Event Graphs. *Communications of the ACM*, 26(11), 1983.

[51] Sally Shlaer and Stephen J. Mellor. *Object Oriented Systems Analysis: Modeling the World in Data.* Prentice Hall, 1988.

[52] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States.* Prentice Hall, 1992.

[53] Francoise F. Soulie, Yves Robert, and Maurice Tchuente, editors. *Automata Networks in Computer Science.* Princeton University Press, 1987.

[54] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling.* MIT Press, 2 edition, 1987.

[55] Stephen Wolfram. *Theory and Applications of Cellular Automata.* World Scientific Publishing, Singapore, 1986. (includes selected papers from 1983 - 1986).

[56] William A. Woods. What's in a Link: Foundations for Semantic Networks. In Daniel Bobrow and Allan Collins, editors, *Representation and Understanding*, pages 35 – 82. Academic Press, 1975.

[57] Bernard P. Zeigler. *Multi-Facetted Modelling and Discrete Event Simulation.* Academic Press, 1984.

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

General Terms: Design, Modeling, System
Additional Key Words and Phrases: Abstraction, Computation

About the Author:

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 B.S. in Mathematics from the Pennsylvania State University, M.S. in Applied Science from the College of William and Mary, and Ph.D. 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. Dr. Fishwick created the `comp.simulation` newsgroup (Simulation Digest) in 1987 and moderated the group until December 1993. He is a senior member of the IEEE and the Society for Computer Simulation. Dr. Fishwick 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*.