SUBMISSION FOR SPECIAL ISSUE ON MODEL SPECIFICATION & REPRESENTATION
FOR ACM TRANSACTIONS ON MODELING AND COMPUTER SIMULATION

# Extending Object-Oriented Design for Physical Modeling

Paul A. Fishwick
Dept. of Computer & Information Science and Engineering
University of Florida
Bldg. CSE, Room 301
Gainesville, FL 32611
*E-mail:* fishwick@cise.ufl.edu
*Phone and FAX:* (352) 392-1414
*WWW:* http://www.cise.ufl.edu/~fishwick

July 12, 1996

# Extending Object-Oriented Design for Physical Modeling

Paul A. Fishwick

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

October 10, 1996

## 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* shown in Fig. 1. 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 re-use. 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

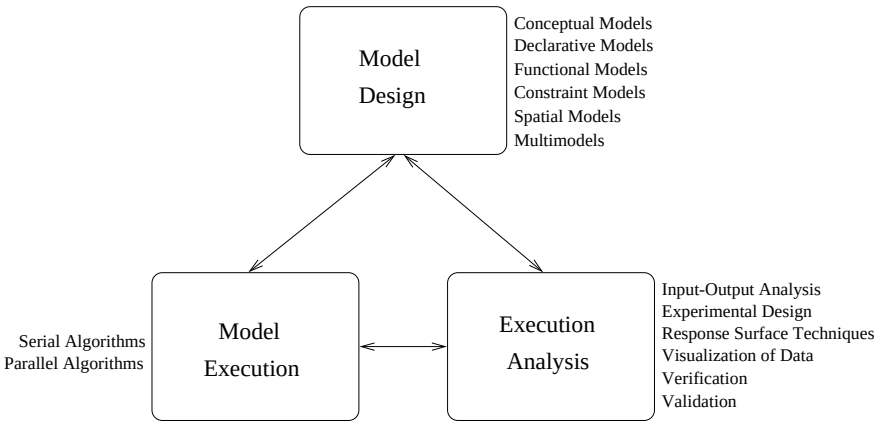



Figure 1: The study of computer simulation: three subfields.

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.

The problems facing most model integration situations are analogous to the carpenter, plumber, and electrician constructing a house. Each has separately created a working network: the carpenter has built the frame, the electrician has soldered wires and boxes for a completely wired house circuit, and the plumber has assembled the plumbing and heating. No blueprint has been used to guide each of the three, and so arguments arise and fast but inelegant fixes are made just to "make it all work." The three work in isolation in performing their functions. The absurdity of this situation is clear. Without a blueprint to serve as a design of house integration, the house cannot be constructed. To model the large-scale systems properly, many models must be written and assembled. We must find a way for the people to communicate by specifing their models using a common framework. After-the-fact glue and paste methods may be necessary initially, but a knowledge representation and design framework must be created. Only then, should computer code be written to execute the models.

In Sec. 2, we discuss our motivation for this work as well as a literature search, specifying what groups have performed similar studies to our own. We introduce a generic example scenario, in Sec. 3, of a robot moving over a space. This example will be used throughout the paper as a common thread. Then, in Sec. 4, we define our concepts as well as object-oriented design as it is practiced in software engineering and simulation. Since modeling is a process, we present the model design aspects for *model engineering*, which defines how models are created from first principles and a knowledge of the system to be modeled. Sec. 4 supplies the phases needed in engineering the model. Sec. 5 discusses the construction of the conceptual model. The conceptual model provides a kind of skeletal structure or scaffolding which requires attributes and methods to make it complete. Sec. 6 describes the modeling approach for satisfying this completion. There are two types of models: static models (Sec. 6.1) and dynamic models (Sec. 6.2). In Secs. 7 and 8, we close the article with

a description of our current implementation of this model engineering approach, MOOSE, and then summarize what we've learned and our future directions.

## 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 [13], 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 [36, 56] 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 programming 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 [35] 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 [54, 56, 39] 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 [16] demonstrated this translation using the DEVS [56] 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 [39]. Object-oriented methodology in simulation has a long history, as with the introduction of the Simula language [3], 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 software engineering

texts [45, 5, 18]. 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 [41] as well as numerous Winter Simulation Conference sessions over the past ten years. Also, various simulation groups have adopted the general object-oriented perspective [44, 57, 22, 1].

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 [45] and Booch [4] and extended these approaches. Regarding the motivation for deriving a new method for dynamic modeling, we offer the following reasons:

1. *Object-Oriented Design:* The new taxonomy is one based on object-oriented design methodology since it is developed as an extension to object-oriented design. Existing object-oriented design for software engineering does not include the concept of modeling. Our design extends the design approaches in software engineering to employ both static and dynamic models for physical objects. Furthermore, we include both static and dynamic models, which capture an integrated physical system with geometry and dynamics.

2. *Orientation*: Even though we describe our method as "object-oriented," the method places equal value on processes and objects. A *process* is surfaced through the use of *dynamic models* (ref. Sec. 6.2), which are stored as method of objects. The problem with traditional object-oriented design is that it tends to bury the process as code while surfacing objects through visually oriented class hierarchies and object relations. It is necessary to bring out both the concepts of process and object by interweaving them: objects contain visual models which, in turn, refer to other objects' attributes and methods, in a chain-like fashion. So, while the approach we advocate is object-oriented in the sense that we organize all our knowledge by developing classes and objects first, *process* has equal footing in the form of visually defined dynamic models.

3. *Completeness*: The new taxonomy organizes models from several different areas including continuous, discrete and combined models under one umbrella. The traditional modeling taxonomies are currently separate. For example, models that require a continuous time slicing type of model execution, are grouped into a model category based on the need for time slicing, not the form of the model. We focus, instead, on visual model structure as a basis for model design organization.

4. *Multimodels:* There does not exist a good modeling approach to multi-level models where levels are defined using heterogeneous model types. The new taxonomy addresses this problem through the multimodel concept.

5. *Design versus Execution:* A taxonomy for modeling should be based on the design of a model and not how it is executed. The current taxonomy introduces some ambiguity as

to whether design or execution is being used to categorize models. The new taxonomy is one which stresses model form as *graphical structure* where possible.

6. *Dynamic Models versus Programs:* We draw a clear parallel between the new taxonomy for dynamic models and programming language categories in computer science. The ability to use the same categories (for modeling as well as for programming) lends credibility to the new taxonomy, and allows one to draw direct parallels between programming language and model design constructs without inventing new modeling category types.

While there has been significant coverage in the simulation literature for analysis methods [2, 25], the general area of modeling for simulation has lacked uniformity and in-depth coverage. Two areas of modeling termed "discrete event" and "continuous" are defined in the simulation literature. For discrete event models, the field is sub-divided into *event-oriented*, *process* and *activity-based* modeling. To choose one of these sub-categories, we might ask "What is an event-oriented model?" There is no clear definition [2, 25] other than to state that a discrete event model is one where discrete events predominate. There is no attempt to further categorize or classify the *form* taken on by an event-oriented model. In mentioning form, we need to address the differences between syntax (form) and semantics (execution). A program or model may be of a particular form; however, the semantics of this form may have a variety of possibilities. A Petri net [38] has a particular form regardless of the way in which it is executed. Ideally, then, we would like to create a model category that classifies the form of the Petri net, apart from its potential execution characteristics. By associating integer delay time with Petri net transitions, one can execute the Petri net using time slicing, discrete event simulation or parallel and distributed simulation. By providing an "event-oriented" model category, it is not clear whether this includes only those models which have explicitly surfaced "events" in their forms (as in event graphs [48] or animation scripts [13]) or whether a GPSS or Simscript program [34] could be considered an event-oriented model. Our approach is to clearly separate model design (syntax) from execution (semantics). Moreover, as stated earlier, programs are not considered to be models at least for most textually-based programming languages. One can attach semantics to syntax, but they remain orthogonal concepts.

Some model types that are similar in form are unfortunately separated into different categories using the traditional terminology. An example of this can be found in block models for automatic control and queuing networks. A functional block model and a queuing network model are identical in form, the only differences being in the semantics for the blocks (i.e., transfer functions) and the nature of the signal flowing through the blocks (discrete or continuous). Our taxonomy stresses a difference in model topology and structure instead of separating model types based on time advance or signal processing features. By using the concept of *functional model*, we characterize the syntactical form of the model: functional models are identified by a uni-directional flow through a network of nodes through directed arcs. In this fashion, control networks and queuing networks are of the same model type. Likewise, this flow is directly analogous to functional composition in functional programming languages.

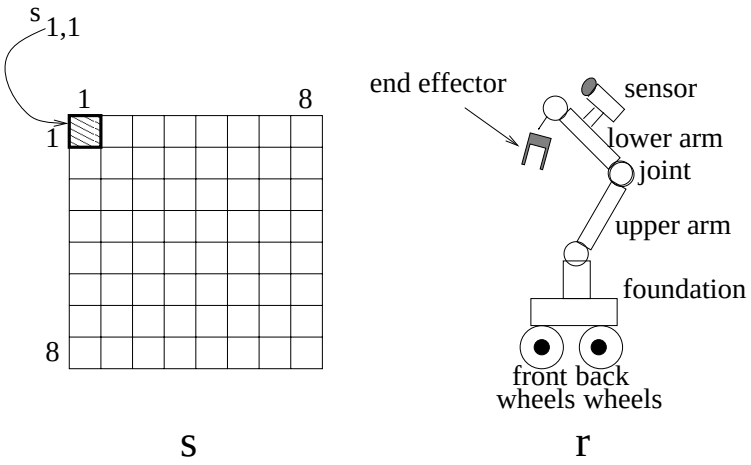



Figure 2: Scenario for generic object behavior.

## 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 (ref. Sec 6.1). A set of mobile robots move around $s$, undergoing change, as well as changing attributes of $s$. Fig. 2 illustrates $s$ and a sample robot $r$. In the remainder of the paper, we will refer to Fig. 2 using classes, objects, attributes and methods defined in Sec. 4. 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.

## 4 Model Engineering

Our basis for physical modeling begins with object-oriented design concepts as described in textbooks [5, 45] as well as object-oriented modeling as applied specifically for simulation of discrete event systems [57]. Model engineering is the process of building static and dynamic models for a physical scenario using our extended object-oriented framework. The steps we take in this procedure are shown in Fig. 3.

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 |

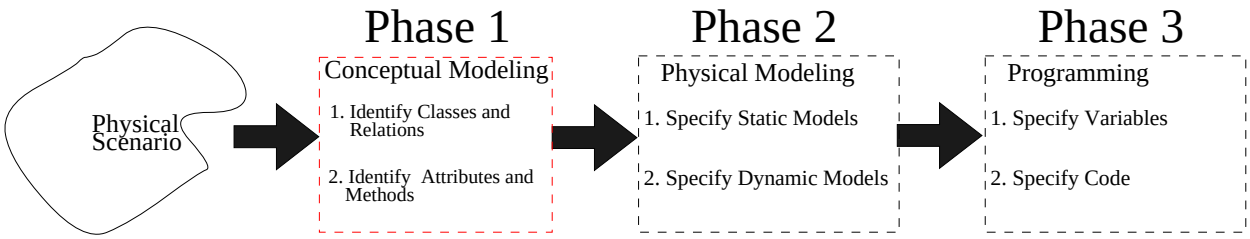



Figure 3: Model engineering.

## 5 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. 4 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. 4 can specify cardinality for

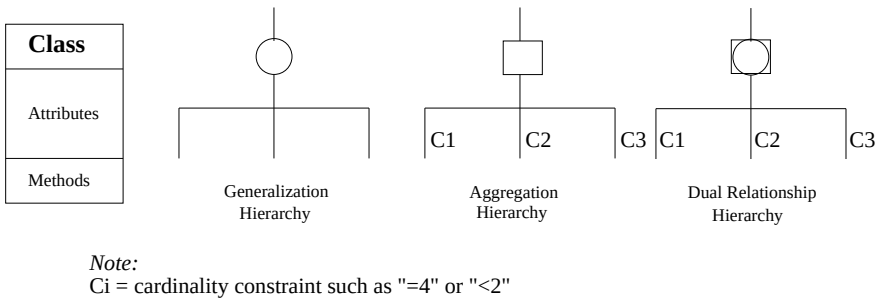



Figure 4: Structure of a class with three relations.

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 [45, 22] 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 [13, 18] 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, sub-classes 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 "inherits" 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

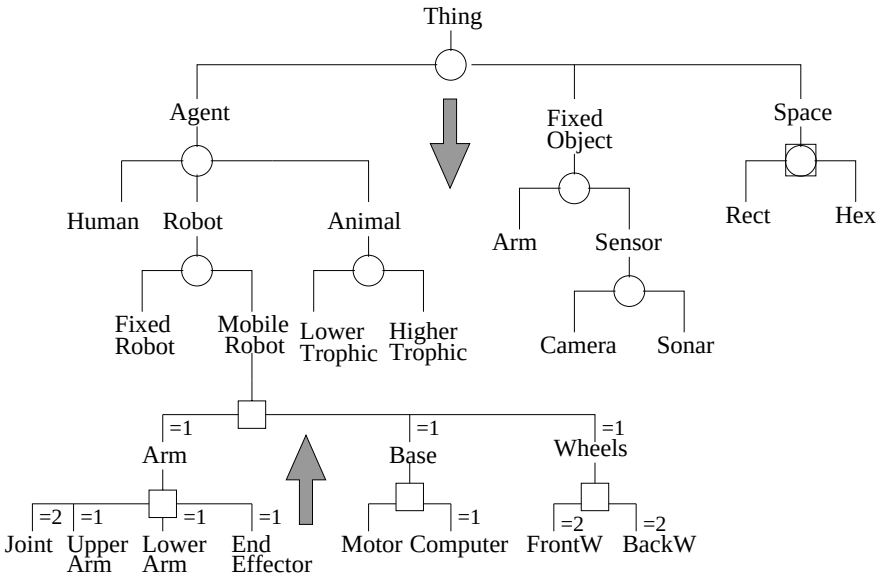



Figure 5: Phase 1, Step 1: Identify classes and relations.

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. 4 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. 2 provides us with the base classes for Fig. 5. The downward and upward block

arrows in Fig. 5 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 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,

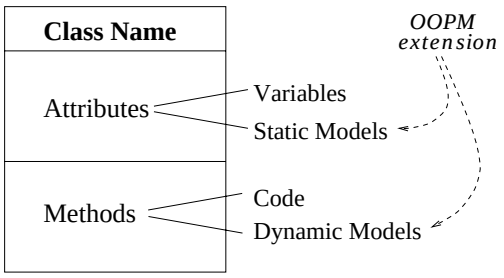



Figure 6: Structure of a Class.

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. 5. 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."

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. 6. Variables and code are described in OO languages such as C++ [49]. 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

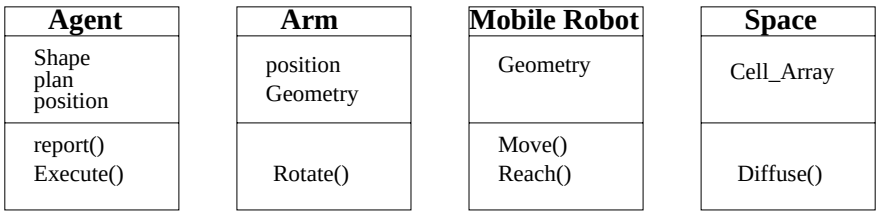



Figure 7: Phase 1, Step 2: Identify attributes and methods.

will use an italics font to differentiate them from surrounding text.

To provide some examples:

- A four-foot high robot *r1* can be represented as: *Mobile_Robot::r1.height = 48.0*, where *height* is specified in inches. Alternatively, we can simply leave out the class name: *r1.height = 48.0*, since we know that *r1* is an instance of *Mobile_Robot* by reviewing the conceptual model. If there is a static model of a robot, in the form of a computer aided design representation, then we would refer to this as *r1.Geometry*.

- A robot arm with a revolute joint will rotate, so we can create *arm.angle* as an attribute of object arm and also *arm.rotate()* as a code method of arm that changes *arm.angle*. If the dynamics of rotation are captured in a dynamic model Rotate, then we have *arm.Rotate()* as my dynamic model. Object *arm* relates to object *r* in that the robot class is an aggregate class containing sub-classes such as *Arm*. Moreover, an attribute inside *r* called *connectivity*, with a linked list structure, would include *arm*. Should the linked list *connectivity* be a static model or a variable of *r*? This is ambiguous and under the user's control. A general heuristic is that if an attribute contains a data structure specifying geometry or relative position of sub-objects, then we call it a model as opposed to a variable, but a case could be made either way.

- A *landscape* is an object that aggregates an array of cells or patches. The dynamics of *r*, that moves within a cell, may be coded as *r.Move()*, a dynamical model or simply as *r.move()*, a piece of code. A landscape subcell, *landscape.cell[2,3]* will contain some number of robot objects using the appropriate data structure.

Fig. 7 illustrates a subset of the objects shown in Fig. 5. For this subset, we identify some attributes and methods that we feel are necessary in simulating the scenario. At this point, it is not necessary to identify the precise structure of each attribute and method since this is part of the conceptual modeling phase. In drawing the attribute and methods, it is useful to recreate Fig. 5 with the *expanded* class nodes shown in Fig. 7. Since this diagram would be large for all classes in Fig. 5, we are illustrating a subset of all class nodes. We use the scheme previously discussed to differentiate models from variables and code: models begin with a capital letter and methods end with a pair of parenthesis "()".

Fig. 8 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 technology," *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.

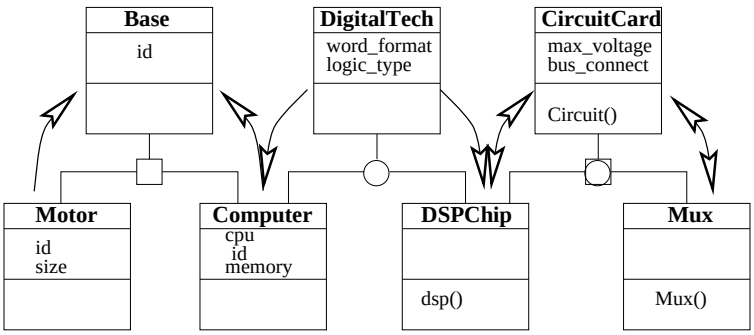



Figure 8: Generic generalization and aggregation scenario.

- 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.

In object-oriented design, there are certain key characteristics that one must employ throughout the design process:

- *Coupling:* In class hierarchies and graphs, classes are coupled together via relations, most of which are aggregation or generalization. Coupling also extends to static (ref. Sec. 6.1) and dynamic (ref. Sec. 6.2) models where objects, attributes and methods are coupled together in graph form to create a model. Coupling provides the glue used to bring classes and other object-oriented features together.

- *Hierarchy:* When a relation can be applied transitively (*Camera* is a kind of *Sensor* that is, in turn, a kind of *Fixed_Object*) then this provides a convenient ordering of knowledge. Furthermore, the transitive generalization and aggregation relations permit the passing of class structure down and up hierarchies. Hierarchy plays a key role in modeling as well with components having sub-components. Sub-component children can be of the same type as the parent component (homogeneous hierarchy) or they can be of different types (heterogeneous hierarchy).

- *Encapsulation:* Where does a particular model belong? To help make this decision, we use the following rule: a model (static or dynamic) is encapsulated within the most specific class or object that contains all attributes and methods defined in that model. This is discussed in detail within Sec. 6.2.4.

Coupling provides the basis for sticking object-oriented components (class, object, attribute, method) together, whereas hierarchy and encapsulation provide ways in which the coupled components can be efficiently managed.

To summarize the conceptual modeling phase, we construct classes and relations among classes. Two key hierarchical, recursive, relations that involve structure passing are generalization and aggregation. The conceptual model exists soley as a knowledge representation of a physical system, and to permit operations such as structure passing and logical inference.

# 6 Physical Modeling

## 6.1 Static Modeling

The word "static" in static models refers to the inability of the model to cause change of attribute; it does not mean that the model doesn't change. For physical modeling, our primary type of static model is one that specifies the topology or geometry of a physical object such as *r*. However, a semantic net [53], would be an equally valid static model. Dynamic models (ref. Sec. 6.2) have the ability to change static models over time. The previously discussed conceptual model, composed of classes and relations, can also be seen as dynamically changing with class relations changing over time. If this occurs then it is logical to create an all-encompassing class called *universe* and then make the conceptual model an attribute of *universe* in the form of a static model.

For modeling geometry and space, there are a number of representational techniques, many of which are discussed by Samet in two volumes [47, 46]. We will not create any extensions of static modeling methods. Instead, for our scenario conceptual model in Fig. 2, we'll discuss our alternatives with an example or two. In Fig. 2 we have two items: a space *s* where robots behave. Space *s* can be modeled a simple square array, which hardly can be classified as a model except that it is a model considering that it is an abstraction of a physical object (i.e., a physical space). Beyond this straightforward model, it is often useful to model space using varying degrees of resolution depending on the area of concern. Areas of space with a sparse density of robots, for example, might be modeled "in the large" whereas dense areas are subdivided hierarchically. A quadtree represents a simple form of four-ply tree data structure that can be used to model the space. For 3D spaces, octrees provide a related structure. Likewise, for the robot depicted in Fig. 2, there are methods

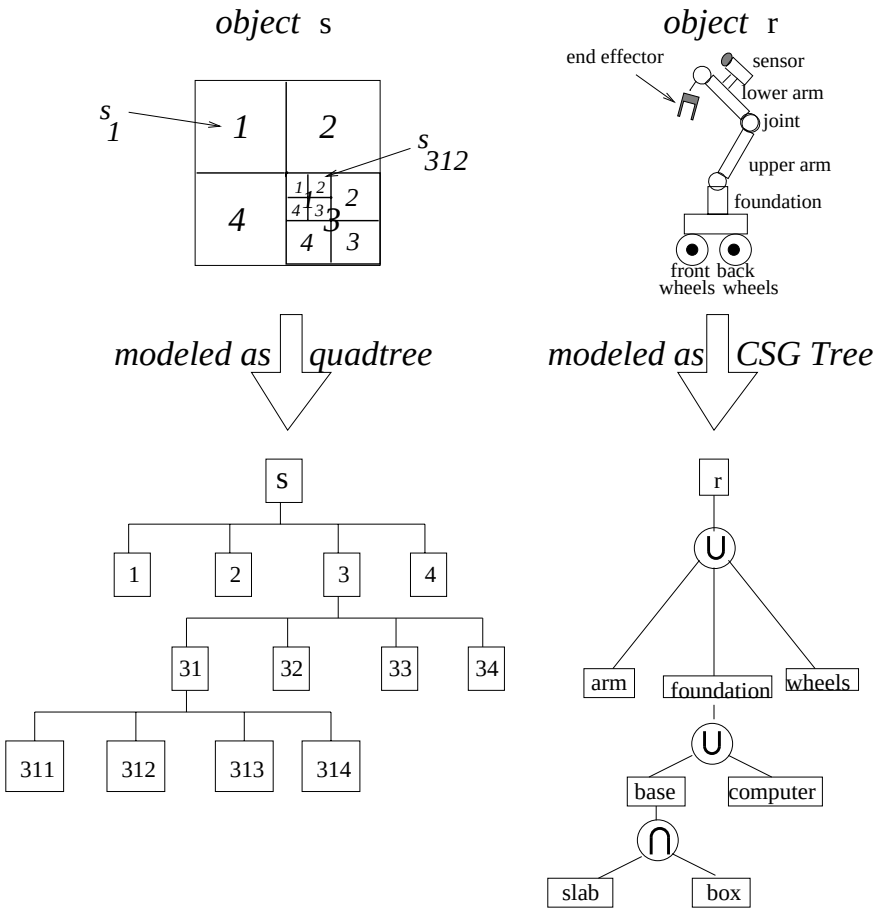
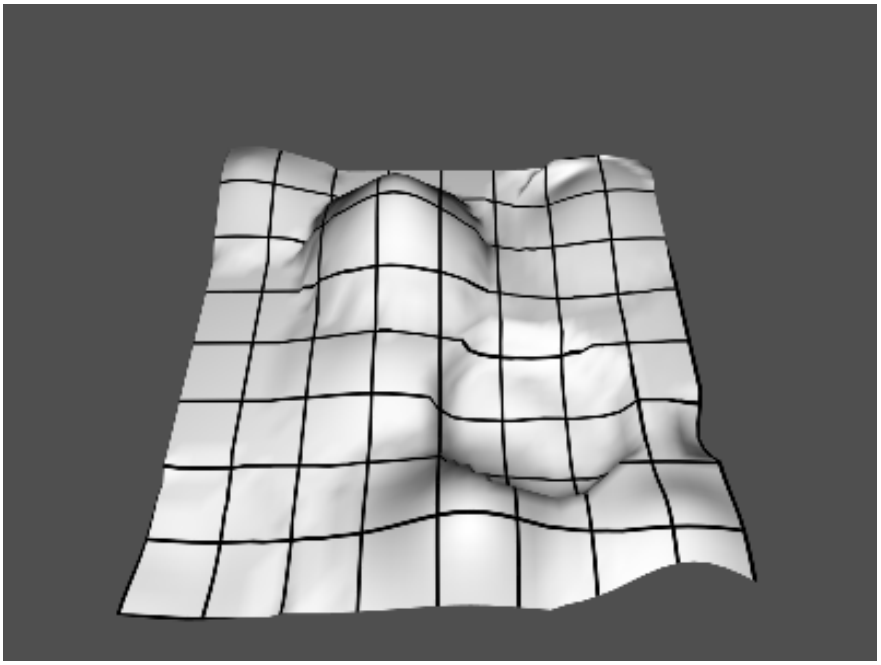



Figure 9: Phase 2, Step 1: Static models for space $s$ and robot $r$.

described in the literature on computer aided design and computer graphics [17, 20]. Fig. 9 displays two static models: the first model is a quadtree of space $s$ and the second model is a constructive solid geometry model of robot $r$.

The quadtree is composed entirely of objects, which when aggregated, form $s$. An object can contain its own static models or an attribute called *contains* that refers to a list structure of robots found in the object. Therefore, $s_1$ and $s_{312}$ are sample objects that may contain several robots. The reason for sub-partitioning a particular cell is that, depending on the density of robots in a particular part of $s$, we may want to sub-divide our space. Another reason is that if $s$ represents a landscape, we may wish to focus our modeling resources in a particular sub-cell within $s$, while still maintaining a partition of $s$. The CSG tree on the right part of Fig. 9 provides a structure for the topology of $r$. The symbol $\bigcup$ denotes union and $\bigcap$ denotes intersection. A CSG tree contains two types of nodes: operations and objects. For example $r$ is an object composed of the union (operation) of objects *arm*, *foundation* and *wheels*. A rectangular *slab*, when intersected with a cubic *box*, creates the *base* of $r$. Figs. 10(a) and 10(b) display 3D geometric static models for both $s$ and $r$, reflecting a more realistic scenario configuration suitable for animation and immersive situations.



(a) Geometry for space $s$.

(b) Geometry for robot $r$.

Figure 10: Geometry representing static models for the scenario.

## 6.2 Dynamic Modeling

### 6.2.1 Overview

A dynamic model captures the way in which attributes change over time. At first glance, it may appear that one can create a dynamic object-oriented model by linking together objects in a graph. For example, supposing that robot $r_1$ gives food to $r_2$. A temptation is to draw an arrow from one object ($r_1$) to the other ($r_2$) and claim this as a dynamic model. In traditional object oriented design, this *message passing* approach is specified as a way to model the "behavior" of objects interacting with one another via messages passed from an object to another. Unfortunately, while such a graph represents our intention of expressing dynamics, it contains no information as to the underlying dynamics. The primary problem is that we do not know which methods of $r_1$ or $r_2$ to use. Suppose that $r_1$ has many potential dynamic models. Which particular dynamic model should we use in our "object" model? The specification of an object pointing to another object is not sufficiently defined to be of any real use for simulation. In effect, a graph containing the two objects is a static model, not a dynamic one, since the graph depicts a geometric or semantic relation: one robot connected to another. To accurately represent the dynamics of objects, we need a more comprehensive and flexible approach that affords the modeler the ability to use familiar models such as FSAs, System Dynamics graphs, compartmental flow models and block models.

### 6.2.2 Three types of Dynamic Model

The three model types that have strong ties with programming languages are: declarative, functional and constraint. A declarative simulation model is one where states and event transitions (individually or in groups) are specified in the model directly. Production rule languages and logic-based languages based on Horn clauses (such as Prolog [24]) create a mirror image of the declarative model for simulation. Moreover, declarative semantics are used to define the interpretation of programming language statements. A functional model is one where there is directionality in flow of a signal (whether discrete or continuous). The flow has a source, several possible sinks, and contains coupled components through which material flows. Functional languages, often based on the lambda calculus [31, 40], are similar in principle. If programming language statements are not viewed declaratively, they usually are defined using functional semantics. The languages Lisp [51] and ML [37] are two example functional languages. Lisp has some declarative features (side effects) whereas other functional languages attempt to be "pure." Finally, with regard to computer science metaphors, constraint languages [6, 29] reflect a way of programming where procedures and declarations are insufficient. The constraint language CLP($\mathcal{R}$) [21] (Constraint Logic Programming) represents this type of language. Also, the next generation Prolog (Prolog III) is constraint oriented. In constraint models, the focus is on a model structure, which involves basic balances of units such as momentum and energy.

Fig. 11 illustrates the dynamic modeling taxonomy. The top level of Fig. 11 refers to the multimodel type (ref. Sec. 6.3) since this type is composed of all sub-types previously discussed by using hierarchical refinement: declarative, functional, constraint and spatial. Conceptual models are generated before multimodels since conceptual models are non-executable and reflect relations among classes. Each of these sub-types has two sub-categories:

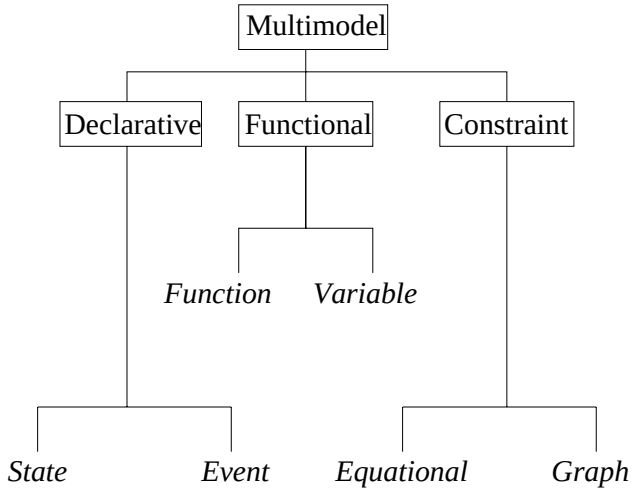



Figure 11: Model taxonomy.

- *Declarative* models focus on patterns associated with states or events. An example declarative model type with a state focus is the finite state automaton. The event graph is an example with the event focus.
- *Functional* models are networks whose main components are either functions (as in block models) or variables (as in the levels found in systems dynamics).
- *Constraint* models are represented as equation sets or as graphs. An example constraint graph is an analog electrical circuit or a bond graph [7].

The extra model type not previously discussed here (but found in Fig. 1 and [13]) is *spatial* model. In our discussion, a spatial model is a static model whose dynamics take on one of three primitive types. So, we do not afford spatial models any special status since in OOPM, they are artifacts of the methodology and of aggregation relations.

### 6.2.3 Definitions

To define how dynamical systems are embedded within OOPM, we need to address some fundamental systems theoretic concepts. A time invariant system $\psi$ can be abstracted as follows: $\psi =< I, O, Q, \Omega, \delta, \lambda >$. The sets $I$ and $O$ represent input and output sets. $Q$ defines the state space for the system. $\Omega$ represents the admissible set of input trajectories, $\delta$ is the state transition function, and $\lambda$ is the output function which is generally a function of $Q$. There are internal and external events. An external event is one from "outside the system" whereas an internal event is one "inside the system" (but from a lower abstraction level). Further explanation and variations of the system formalism can be found in the systems [36] and simulation [55] literature; however, the above definition suffices for our purpose. The first key observation of OOPM is that we are encapsulating behavior (dynamic models) and structure (static models) within objects. This represents a structured representation of a system as opposed to an all encompassing system definition with a multi-dimensional state space spanning many objects. For the aggregate objects, however, this large state space is

accurately captured since these accrue fewer-dimensional state spaces of sub-objects through *composition*. The following definitions are presented to bridge the gap between the formal system components and our components. A class $C_i$ is defined as a sub-class of $C$ in an aggregation relation; likewise, a class $C^i$ is defined as an aggregate object composing $C$ as a sub-class. We let $\hat{C}^i = C \cup C^i$ and $\hat{C}_i = C \cup C_i$ for notational convenience. A similar notation can be constructed for objects $O$.

- *State: A state of class $C$ is contained within $C$'s attribute list.* Comments: objects that are leaves in an aggregation hierarchy will have low-dimensional state spaces with internal and root nodes aggregating these low dimensional spaces. The state of object *arm* for the robot may include a *position* for the centroid as well as an orientation $\theta$. The arm contains all states in $Arm_i$ through composition.

- *Event: An event is classified as internal or external.*

  1. *An internal event for $C$ is an input from within $\hat{C}_i$.* Comments: all internal events within $C$ are inputs from $C$ or sub-classes of $C$. A clock has an internal event when the alarm rings, but the alarm mechanism is a sub-class $C_i$ of $C$ and the event is an output from $C_i$. The computer for robot $r$ is a sub-object of the robot. The computer may periodically produce internal (relative to $r$) events that are employed in a dynamical model in $r$).
  2. *An external event for class $C$ is an input from a class in $C^i$.* Comments: An external event to an alarm clock comes from the human hand or finger object that presses a button to stop the alarm. All environmental activities in $s$ affect $r$ through external events, since the environment is outside of $r$.

- *Input: All inputs to $C$ are either internal or external events.*

- *Output: All outputs from $C$ are inputs to some $P$ with the exception of a sink node, for which an output employs a method of $C$ using the $C$'s state attributes.*

These definitions will help us in formulating common templates for dynamical models. A primitive dynamical model is of three types: declarative, functional or constraint. For each of these model categories, there are some common types that we define with templates.

1. Declarative: A finite state automaton (FSA) is defined with nodes and arcs. A node is a state of $C$, and therefore an attribute of $C$. The arc contains a boolean-valued method $p()$ with arguments that are internal or external events. An event graph is defined with nodes and arcs, as for an FSA. A node is an event (internal or external). An external event relative to $C$ is an attribute of $C^i$ and an internal event is an attribute of $\hat{C}_i$. The arc in an event graph represents a method that schedules or causes the event on the head of the arc. The robot $r$ may be in one of three states: *active*, *stationary* or *maintenance*. The *maintenance* phase is used for fixing the robot's position via a satellite. Therefore, a method is created in $r$ that points to a dynamic model: an FSA with three states. Arcs from one state to another in an FSA are boolean predicates.

2. Functional: A block model has nodes and arcs. A node is a method of $C$, and the arc represents a *directed* connection from one method to another. Both methods can be found in $\hat{C}_i$. The block model is function-based since functions are made explicit as nodes. Variable-based models such as System Dynamics [42] or compartmental models [23] are the duals of function-based models since variables are placed at the nodes. For a $C$ with this type of method, the variables are attributes of $\hat{C}_i$. Functions are often assumed to be linear, but if they are defined, they are methods found in $\hat{C}_i$. For our scenario we would have to derive a directional activity to use a functional model. One such activity is the movement of water on $s$ over cells in a specific direction, such as via a canal or along a river. Such boundary conditions suggest a functional model, since without these boundaries, we would use a constraint model defined by dynamics incorporating local conservation laws. Another activity is if robots cooperate with each other to form a pipeline task, passing a product or food from one end of the robot "chain" to the other. This activity is common in workstation cells in a manufacturing scenario where robots are often fixed relative to each other in space.

3. Constraint: Constraint models are equational in nature, and reflect non-directional constraints among object attributes. A class $C$ with a constraint equational model contains an equation with terms containing attributes of $\hat{C}_i$. Equations can be represented in graph form as well as with bond graphs [7, 43, 50]. Models of non-directional behavior, such as general hydrodynamic models are constraint-based. If all robots interact with each other in ways dictated by nearest-neighbor conditions, for example, this is modeled as a constraint since there is no consistent, time-invariant direction associated with the dynamics.

### 6.2.4 Representing Dynamic Models

How are dynamic model components represented in the physical modeling methodology? We will illustrate two model types (functional and declarative), each with two model sub-types. For the functional model types, we use a block model and a System Dynamics model. For the declarative model type, we will use an FSA and a Petri net. The following notation, consistent with the previous notations, will be used throughout this discussion:

- *Objects:* An object is represented as $obj$. $obj_i$ represents a sub-object of $obj$ (in its aggregation hierarchy), and $obj^i$ represents a super-object which is composed, in part, of $obj$. When indices $i$ and $j$ are used, it is possible that $i = j$ or $i \neq j$. This relation rests with the particular application.

- *Attributes:* $obj.a$ represents a variable attribute and $obj.A$ represents a static model attribute. $a$ is short for any string beginning with a lower case letter; $A$ is short for any string beginning with an upper case letter. Attribute references (i.e. names) and values are relevant: a name is just $obj.a$ whereas the value of attribute $a$ is denoted as $v(obj.a)$. The following special attributes are defined for all objects: $obj.input$, $obj.output$ and $obj.state$ and represent the input, output and state of an object at the current time.

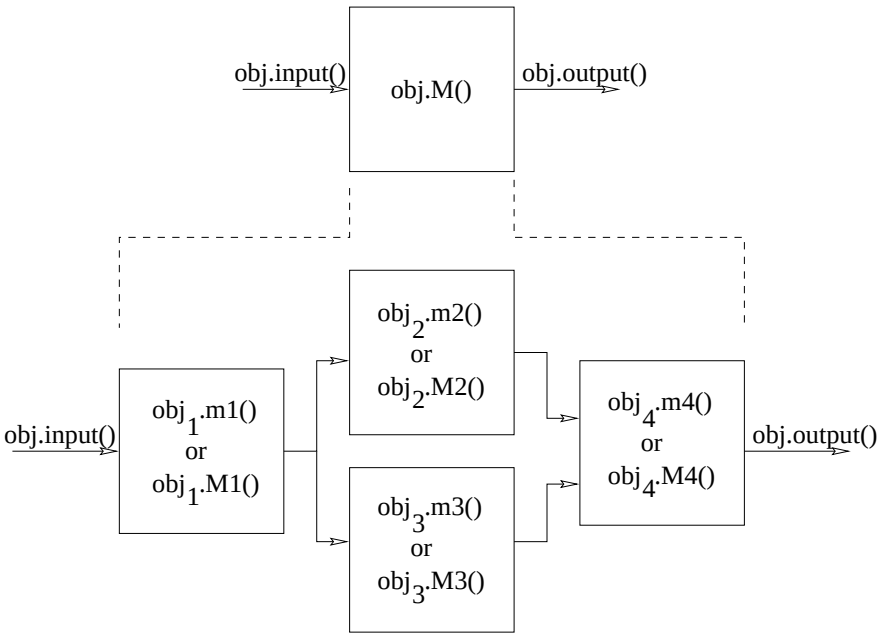



Figure 12: A block multimodel refinement.

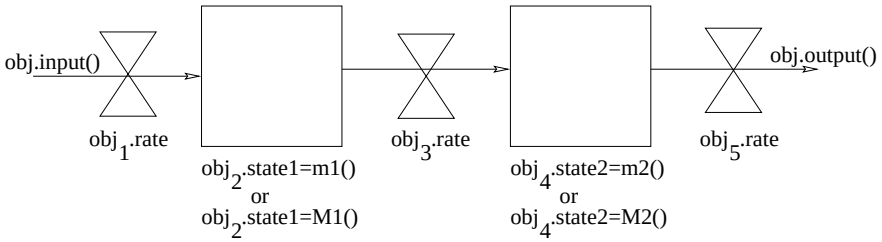



Figure 13: A System Dynamics model.

- *Methods:* $obj.m()$ represents a code method and $obj.M()$ a dynamic model method. $m$ is short for any string beginning with a lower case letter; $M$ is short for any string beginning with an upper case letter. The following special methods are defined for all objects: $obj.input()$ and $obj.output()$ and represent the input and output time trajectories.

The block model contains acollection of blocks coupled together in a network. Each block calculates its input, performs the method representing that block, and produces output. A block is not an object since it represents a method *within an object*. Without any refinement within a block, a function takes no time to simulate. Time is ultimately associated with state change. All $obj_i$ represent sub-objects of $obj$. Fig. 12 displays two simple block models: the first has one block and the second has 4 coupled blocks. The essence of multimodeling is seen here: a method defined as a dynamic model of the same model sub-type: *block*. Fig. 13 shows a System Dynamics model that is similar to the block model except that instead of methods represented by the network nodes, a node represents an object's state (a variable attribute). The same kind of multimodeling represented in Fig. 12 (refining a block into a dynamic model) can be done for the model in Fig. 13; a block is refined into a System

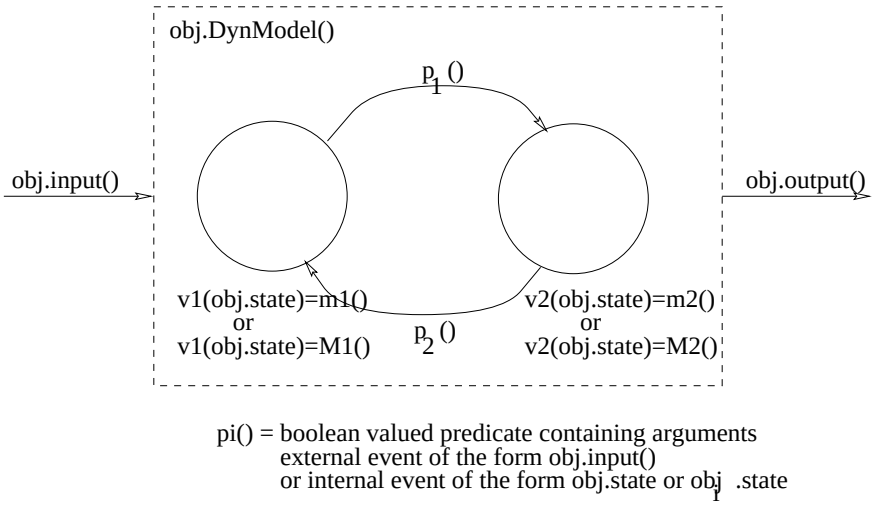



Figure 14: A finite state automaton.

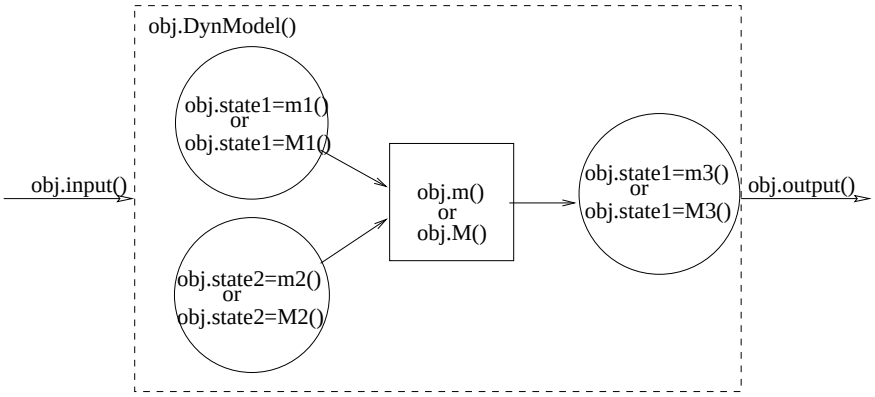



Figure 15: A Petri net.

Dynamics model. This multimodel refinement is particularly useful for our two declarative model types shown in Figs 14 and 15 where *input*() and *output*() are not as obvious unless we capture the model inside of a functional block. Multimodeling is denoted by drawing a dashed functional block around the model, denoting model refinement. The methods *input*() and *output*() are essential to perform *coupling* of models together. An FSA will have an *input*() method and a state variable attribute *obj.state*, but we require coupling information somewhere if we are to decide where the input is coming from and where the output is going to. This coupling information for a method of *obj* is present in some $obj^i$. For example, if the FSA belongs in a robot *r*, defining the dynamics of state change for *r*, the input must come from a physical object outside of *r* but within a more encapsulating object such as the environment.

The predicates $p_1$ and $p_2$ in the FSA model in Fig. 14 require further explanation. A predicate is defined as a logical proposition, whose value is boolean, containing internal and external events. External events are of the form *obj.input*() and internal events are of the form *obj.state* or $obj_i.state$. Rules are another convenient dynamic model (of the declarative type) that express changes of state and event. A rule-based model is composed of a collection

of rules within an object *obj*, each rule $i$ of the form: IF $p_i(event, state)$ THEN $obj.mi()$ or $obj.Mi()$. The phrase $p_i(event, state)$ defines the same logical proposition discussed for the FSA.

### 6.2.5 Single Object

Single objects are our starting point for defining dynamic models. For example, one may create a simple declarative model in object $r$ of class *MobileRobot* by specifying a finite state automaton (FSA) model and linking this to *r.Move()*. We will let the FSA be defined by two states, $S$ and $M$ for "Stationary" and "Move" respectively. A robot is in state $S$ until it receives a cue (an input signal) to move to state $M$. When the robot receives another input signal, it moves back to state $S$. The state variable is located as an attribute *r.position*. Therefore the semantics of the FSA would be such that attribute *r.position* would be modified depending on the state $S$ or $M$. For a single object, expressing the dynamical model in terms of object-oriented features turns out to be straightforward. For a number of objects, the procedure is more involved but still reflects a natural method of modeling.

A single object may be an entity such as $r$ or a more general space such as $s$. For $s$, we define dynamic models to be those that change attributes of $s$. Models such as partial differential equations (PDEs) and cellular automata (CA) can be defined within $s$. For example, the constraint reaction-diffusion model

$$\frac{\partial \phi}{\partial t} = f(\phi) + \nabla^2 \phi \tag{1}$$

as a method (dynamic model) of $s$ describes how $\phi$ changes over time. For our robot scenario, $\phi$ is an attribute of the landscape $s$ that varies over each cell—such as water depth or vegetation density. Furthermore, $r$ can interact with $s$ since a component in a method of $r$ can depend upon cell attributes, and vice versa.

### 6.2.6 $k$ Non-Interacting Objects

Non-interacting objects do not require any additional dynamic models other than the ones encapsulated within each object. If many $O_i$ reflect robots in motion, there is no need for a dynamic model in an $O$ to orchestrate this motion, unless there are constraints placed on the dynamics, in which case a model would be necessary. All $O_i$, for example, would have to be scheduled by a simulation engine so that they could perform their individual tasks; however, this scheduling is a part of the model execution and does not affect model design.

### 6.2.7 $k$ Interacting Objects

- $k$ *directionally interacting objects*: interacting objects with a directional flow use functional models: data of some type flow through the methods of objects to achieve a change of state. For example, assume that two robot objects $r_1$ and $r_2$ are in a quadtree sub-cell called *cell*. $r_1$ always hands a part to $r_2$. This describes the interacting dynamics of an aggregate object $r$ containing $r_1$ and $r_2$. A method $f_1$ is defined in $r_1$ that captures what is done with the part when $r_1$ receives it. Similarly, $r_2$ does $f_2$ to the incoming part it gets from $r_1$. We now make a functional model that is

composed of two coupled functions $f_1$ and $f_2$ ($f_1$ is linked directionally to $f_2$): $f_1 \rightarrow f_2$. This coupling is performed in accordance with the aggregation rule to define how the coupling is to occur. The aggregate functional model becomes a method of $r$.

- $k$ *non-directionally interacting objects*: interacting objects with no directionality in the flow are represented using constraint models. If $k$ objects interact in this fashion, a constraint model is encapsulated within the aggregate object containing all $k$ objects. Robot $r_1$ may interact with $r_2$ in such a fashion. A simple method of interaction is one using a spring metaphor. We attach a virtual spring between $r_1$ and $r_2$. Then, given an acceleration to $r_1$, both robots will move in accordance with Hooke's law in an effort to achieve equilibrium. The equation for Hooke's law is a constraint model located in an aggregate object $r$ that contains both $r_1$ and $r_2$. The equation has terms that are attributes of $r_1$ and $r_2$. The way in which the equation terms are clustered is determined by an aggregation rule for equation construction.

One theme that arises from our methodology of dynamic models is that one locates a dynamic model in the object for which it is appropriate. Often, this object already exists as does $r$ which has a functional model referencing methods stored in $r$'s sub-objects such as *arm*, *end-effector* and *foundation*. However, in some cases, an aggregate object must be created to locate the model, as with the two robots interacting in spring-like manner: the model of their interaction does not belong to either of them, only to their aggregate object. The latter concept also applies to *populations* of objects where they are present in a scenario. To take an example, let's consider a constraint model in equation form. Where do the equation and the composite equation terms belong? If a term or part of an equation contains attributes only of class $C$ then, it belongs in $C$. If a dynamic model in $r_1$ (on the end of a spring) contains only terms of $r_1$'s position (including input terms dependent on time alone) then the model belongs inside $r_1$. If terms including $r_2$'s position are in a term or equation, then we must move the term or equation up a level to be located in $r$.

### 6.2.8 Continuum Models

It may be readily apparent how to take the average scenario object and define it to be object-oriented; however, models that have a *fluid* or non-discrete nature seem to present an incongruity within our defined approach. However, continuum physical phenomena such as fluids, whether gaseous or liquid, need not present a problem for the methodology. If the fluid object is sub-divided into constituent objects in the same way that scalar and vector fields are discretized for numerical reasons (to solve the field equations), then each discrete part is captured as a sub-object of the field. We briefly considered the concept of a river over our robot space $s$. If the river has rigid boundaries, the sub-objects of $s$ will contain sub-objects of the river object. These object aggregate relations are time-dependent. Objects can be seen to move with the field or stay fixed. If they stay fixed, the dynamics associated with each object follow the methods of finite difference formulations. If they move, they become fluid *particle* objects and are not unlike the robots. If a fluid object is compressible, or the object can add or subtract sub-objects, we add these objects over time and our static models change to accommodate the change in structure. This is a case of having to delete

and add objects dynamically. In the following section, we'll treat an example of dynamic object creation using a biological metaphor.

### 6.2.9 Morphogenesis

In the previous examples of dynamic systems, a dynamic model was specified within an object for which it was relevant. But, what if the dynamics cause a change in the static model of an object? This is what happens in biological systems and what we call morphogenesis [33]—a temporal change in structure. Lindenmeyer Systems [30] (L-Systems) capture a dynamic way of modeling that falls under the declarative class of dynamic models: rules are specified to model change of object structure. We begin with an object whose state is defined by an attribute serving as an initial condition $\omega$, and continue simulation by growing a static model tree composed of objects. For most engineered devices, such as robots, we do not generally consider dynamic growth to apply; we apply growth to natural objects. However, we stated earlier that our robot was capable of changing shape, so we can carry this metaphor further by stating that the robot link structure in Fig. 2 is the piece of the robot that grows like a tree over time. At the end of the recursive subdivision, an end-effector grows onto the end of each link chain much as a flower grows at the tip of a branch or plant stem. The recursion defined in the productions provides for a tree of objects that grows and is constructed as the methods are applied. After a tree has grown, other state updating methods can be applied for modifying the object states; however, in the majority of cases, growth and decay methods will continue to be applied in parallel with methods that, for instance, change the state of a tree structure (engineered, biological or otherwise) as a result of external forces such as wind or water.

For our models, we will ignore the *joint* object and not directly model the time-dependent change of link width and length. The L-System production model for simulating the link-tree in Fig. 16 is defined below:

$$
\begin{aligned}
p_1 &: \quad \omega : a \\
p_2 &: \quad a \rightarrow I[A][A]A \\
p_3 &: \quad A \rightarrow IB \\
p_4 &: \quad B \rightarrow [C][C] \\
p_5 &: \quad C \rightarrow ID \\
p_6 &: \quad D \rightarrow K
\end{aligned}
$$

Production $p_1$ is the initial condition (or terminating condition). $p_2$ provides the basic support structure for the plant with an internode, two angle branches and a straight branch. Each branch is constructed with $p_3$. $p_4$ provides a two branch structure at the end of each of the three branches just created, and each of these new branches contains a flower (via productions $p_5$ and $p_6$). In the case of L-System model definition, productions $p_1$ through $p_6$ represent a single dynamical model defined as a method of the overall aggregate robot object $r$; however, it is also possible to separate rules so that a rule defines the dynamics for the object for which it applies. In this case, the rule becomes a method in that object. This

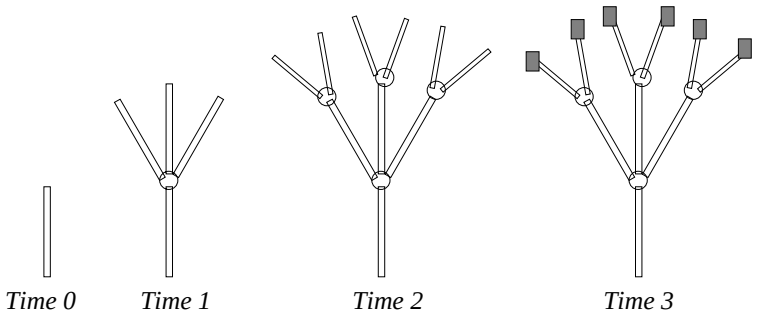



Figure 16: Time snapshots for robot link tree.

"distributed" rule approach is a logical one and since $r$ represents an aggregation of all sub-objects, all six rules are aggregated as a method of $r$ through the composition properties of aggregation. These approaches may suggest different model execution methods for globally situated rules and distributed rules, but we shall not address the model execution issues here since our focus is on model design.

We let time advance be associated with the execution of rules until the appearance of a physical segment ($I$ or $K$) occurs. The changing state of the robot link tree is shown in Fig. 16. The static model for the growth at time 3 is illustrated in Fig. 17 and the corresponding object and sub-object definitions are illustrated in Fig. 18. Fig. 18 also contains the "internode" $I$ objects that define the individual links.

### 6.3 Multimodels

We identified dynamic models as being one of three types, and it is possible to create a hierarchy of dynamic models by refining a component of one model as representing another dynamic model. So, for example, one may take a state of $r$ and refine this into a functional model containing two coupled functions. This sort of model decomposition is called heterogeneous model decomposition [11, 32, 13] since more than one model type is used during refinement. Homogeneous refinements are more commonly used, where a model component is refined into similar components but using more detail. In [13], multimodels were visualized outside of an object-oriented framework. In OOPM, a multimodel may be embedded in several physical objects; however, the individual multimodel layers can still be abstracted by refining dynamic model components. Even though we have specified multimodels as applying to dynamic objects, their utility is just as applicable to static models. For example, consider a static model of $s$: object $s$ contains a quadtree model as an attribute. Each cell of the quadtree contains static models of all robots $r$ inhabiting the cell. Moreover, a link contained with $r$ may be subdefined into yet another model type: a collection of finite volume objects used mostly in finite element analysis.

Multimodels, whether of static or dynamic models, involve changes in scale so that as we refine our models, we change the scale of our scenario and new sub-objects emerge at each abstraction level. For homogeneous refinement, a scale change is accompanied by a regular kind of scale change. Consider the static case first. A landscape $s$ at one level can be sub-divided into cells that are the same shape as $s$ but have metric transformation applied to each of them. This represents a model type we will call *array*. For dynamic models, one

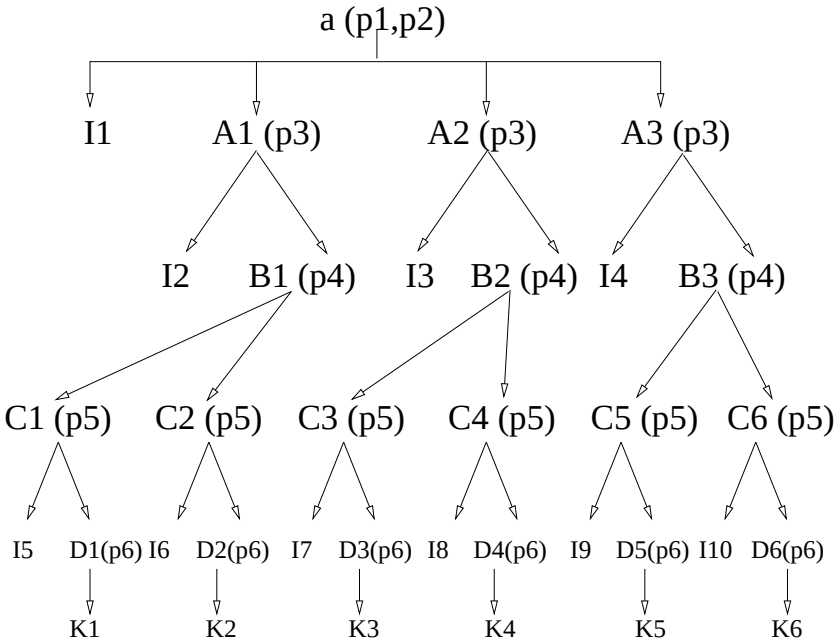


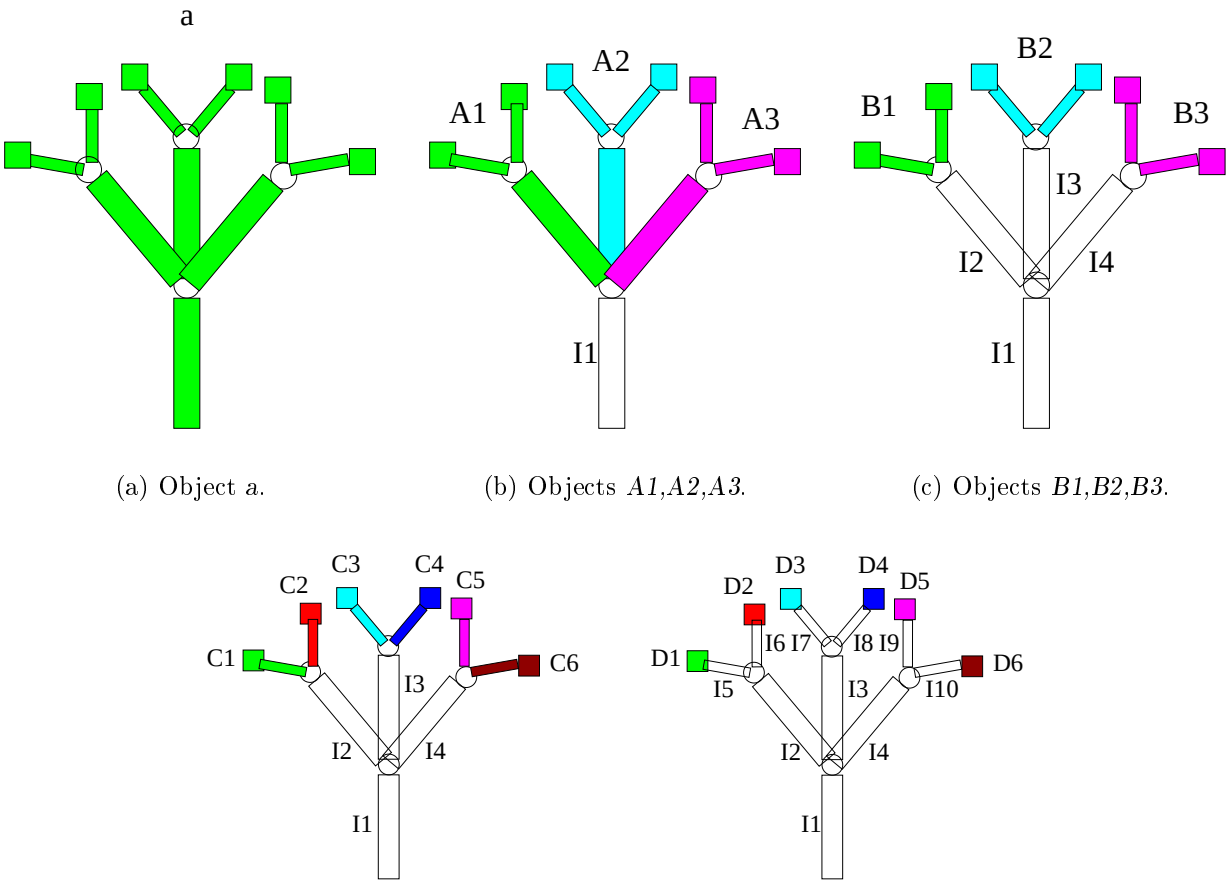

Figure 17: Static model for the robot at time 3.

can look inside $r$'s computer to find a digital design, composed of interconnected blocks. Each block is subdivided into blocks, yet again, but with blocks of finer granularity. On the other hand, with dynamic model heterogeneous decomposition, we find that we define more coarse grained dynamics for $r$ using an FSA with finer grained dynamics using other model types such as the functional block model. For static models, we may decide to subdivide each cell of $s$ using quadtrees. This represents a shift in model type: from an array to a quadtree. Recent work on multimodeling and a new taxonomy for structural and behavioral abstraction is found in [15, 28].

Every dynamic model $obj.M()$ has model components. For multimodeling, the following three model components are important: 1) attribute reference, 2) attribute value, and 3) method. Refinements can be made for each of these model component types.

1. An *attribute reference* is denoted by referring to an attribute $obj.a$. In particular, attributes which hold the set or subset of state space can be used for multimodels by refining the attribute $obj.state$ into a method: $obj.state = m1()$ for a code method refinement or $obj.state = M1()$ for a dynamic model refinement. Examples of this type of refinement are: 1) a Petri net place, 2) a compartment of a compartmental model or 3) a level in a System Dynamics model.

2. An *attribute value* is denoted by referring to the value of an attribute $obj.a$, denoted




(a) Object *a*. (b) Objects *A1,A2,A3*. (c) Objects *B1,B2,B3*.

(d) Objects *C1...C6*. (e) Objects *D1...D6*.

Figure 18: Objects created by time 3.

as $v(obj.a)$. Attribute values appropriate for multimodels reflect a *phase* or aggregate state. A multimodel refinement of an attribute value is performed as either $v(obj.state) = m1()$ or $v(obj.state) = M1()$. Examples of this type of refinement are: 1) an FSA state or a 2) Markov model state.

3. A *method* is denoted by referring to an objects method: either $m1()$ for a code method, or $M1()$ for a dynamic model method. Examples of this are: 1) a function in a block model, 2) a Petri net transition, or 3) a component in a graphical constraint model such as a bond graph. Recall that methods can be constraint relations as well as functions.

### 6.4 Location of Models

In what object or class does a model belong? This is a key question that arises when building the conceptual model. As we have seen with the different dynamic model types, there are specific approaches for model location depending on the type. In general a good heuristic for

model positioning is to *place a model in an object whose composite objects contain all model components*. For example, when we built a functional model for two robots $r_1$ and $r_2$, we put the coupled two-function model of $f_1 \rightarrow f_2$ inside of an aggregate object $r$ containing $r_1$ and $r_2$. The model could not belong in any single robot even though the model's components are located in individual robots. The same heuristic can be employed for all model types. For an equational model, we might have the following model inside object $o1$:

$$\frac{d}{dt}\rho = k_1\rho + u(t)$$

where $\rho$ is a density attribute within object $o1$. Since this model includes only object $o1$'s attribute and reference to an external input, it belongs in $o1$. Contrast this against:

$$\frac{d}{dt}\rho = k_1\rho + o2.\gamma$$

which contains a constraint relation including a term involving another object's ($o2$) attribute $\gamma$. This model must be placed in an aggregate object that contains both $o1$ and $o2$.

### 6.5 Predator-Prey Model

Consider that some robots act like predators and some act like prey. In this case, an applicable dynamic model to create is along the lines of the Lotka-Volterra model [33]. A conceptual model is shown in Fig. 19. This model suggests that we have a physical scenario composed of an environment (weather), landscape and a population of organisms. There are two types of populations: predator and prey. For the sake of the biological metaphor, we choose *Panther* as the class of predator and *Bird* and *Deer* as sample prey classes. The Lotka-Volterra model is an example of a general population model that can be characterized as a *p-state* ecological model [10]. The designation of *p-state* is positioned in Fig. 19 within the *Population* class where it belongs. For completeness, we have included other types of ecological models [10, 19] and where they fit within the class hierarchy:

- General Population Model (*p-state*): a model that specifies the dynamics of single or inter-species populations.
- Structured Population Model (*i-state distribution*): a population model where other independent variables such as size or age are used to "structure" the population into classes. We placed this within class *BirdPop*. A discrete set of structured classes could also be created under *BirdPop* if desired, such as *Hatchling*, *Juvenile* and *Adult*.
- Individually Model (*i-state configuration*): a set of continuous-time equations, one per individual. If one chooses a discrete event-type approach, using rules for the model type for example, other model types are possible. Wolff [52] refers to a rule-based model as an individual-oriented model (IOM) to differentiate it from the i-state configuration model, termed an individual-based model (IBM).

Let's note the rules for generalization and aggregation:

- Aggregation:

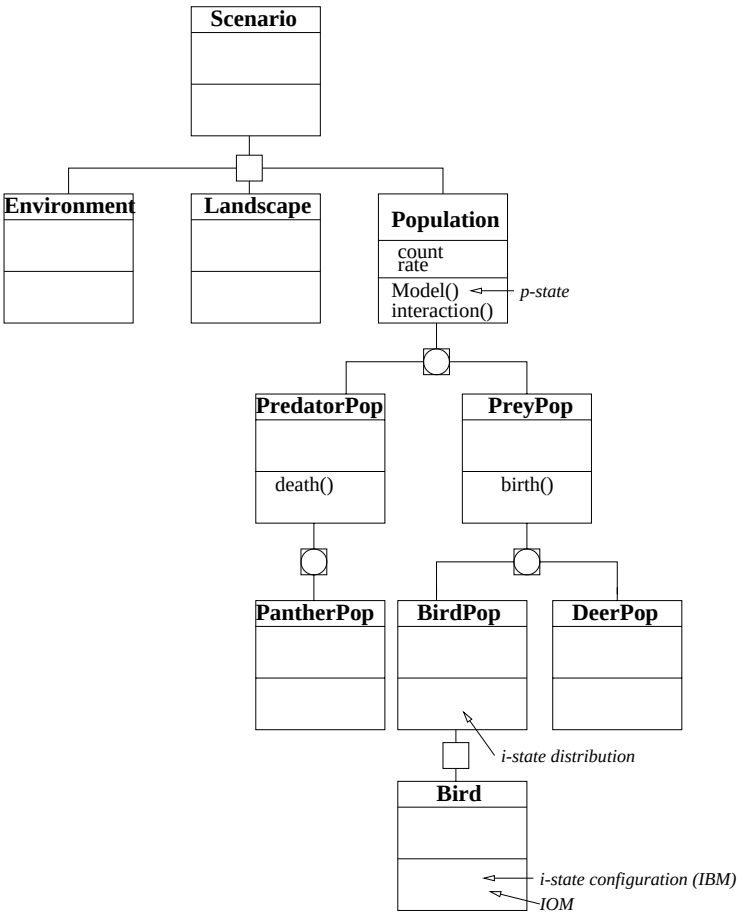



Figure 19: Lotka-Volterra population dynamics.

1. An specific aggregation rule for attribute *count*:

$$Population.count \quad = \quad PredatorPop.count + PreyPop.count$$

   A more general aggregation rule for *count*, keeping in mind updates and additions to this conceptual model, is:

$$C.count \quad = \quad \sum_i C_i.count$$

   where $C$ matches any class containing *count* and $C_i$ matches the sub-classes of $C$.

2. An aggregation rule for dynamic model method *Model()* in *Population*:

$$\begin{aligned} PredatorPop.rate &= PreyPop.birth() - PredatorPop.death() \\ PreyPop.rate &= PreyPop.birth() - interaction() \end{aligned}$$

3. An aggregation rule for code method *interaction()* in *Population*:

$$interaction() \quad = \quad PreyPop.count \times PredatorPop.count$$

- Generalization: We let *count* be inherited (passed down) but not any of the methods defined in *Population*, *PredatorPop* or *PreyPop*.

In reviewing our model, we realize several important benefits from the use of OOPM in creating this model. The main benefit is one of knowledge representation that focuses on class creation and the lexical naming of equational terms such as *birth()* and *interaction()*. By making names explicit, we make the model more comprehensible. The benefits of structure passing are inheritance and aggregation. The definition of *Population.Model()* is invariant to additional *PredatorPop* or *PreyPop* sub-classes we may choose to add in the future. For example, we may later add *AlligatorPop* under *PredatorPop*. Since *AlligatorPop* would pass *count* upward via the first aggregation rule, the population model need not be redefined.

## 7 Programming and Implementation

We are building a system called MOOSE: multimodeling object-oriented simulation environment. MOOSE will capture the essence of OOPM by leading the user through the phases of model development shown in Fig. 3. The user constructs a conceptual model using a Tk/Tcl graphical user interface. The user adds attributes and methods. For those attributes and methods that are defined as *models*, there is a model window that permits visual editing. As the model is executed, the simulation output is shown on a scenario window. Our progress to date has illustrated the use of simulation to the planning process [27, 26]; however, we are still building the graphical user interface utilities for the model window. A sample scenario window for an air force mission application was constructed along with a simulation built on top of SimPack [12, 13, 9].

We draw a dividing line between the actual implementation and the *logical design* which is used as a basis for code implementation. Our focus in this article has been on this logical design. Some implementation choices, for example, have dictated that for a particular object-oriented language, creating formal objects for every individual in a population may not be computationally feasible, even though our design is drawn so that these objects exist. This is not a problem and reflects that the design stands by itself and is used as an intermediate vehicle from concept to code. Different computer languages have their own unique features, and the basic object-oriented physical design should not be bound by what is offered or not offered by these languages. C++ is an example of a language that offers inheritance but not composition so while inheritance is supported in the form of derived class structure, composition is handled on a case-by-case basis where the programmer stores the structure in the object(s) affording computational efficiency.

## 8 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 [8, 14], 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 engineering 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. In particular, Brian and Robert critiqued an early version of the manuscript and Tolga designed the geometry in Figs. 10(a) and 10(b). 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


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

[2] Jerry Banks and John S. Carson. *Discrete Event System Simulation.* Prentice Hall, 1984.

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

[4] 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.

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

[6] Alan H. Borning. THINGLAB – A Constraint-Oriented Simulation Laboratory. Technical report, Xerox PARC, 1979.

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

[8] Timothy Budd. *An Introduction to Object Oriented Programming.* Addison-Wesley, 1991.

[9] Robert M. Cubert and Paul A. Fishwick. OOSIM User's Manual. Technical report, University of Florida, Department of Computer and Information Science and Engineering, 1996.

[10] Donald L. DeAngelis and K. A. Rose. Which Individual-Based Approach is Most Appropriate For a Given Problem? In Donald L. DeAngelis and Louis J. Gross, editors, *Individual-Based Models and Approaches in Ecology*, pages 67–87. Chapman and Hall, New York, 1992.

[11] Paul A. Fishwick. Heterogeneous Decomposition and Coupling for Combined Modeling. In *1991 Winter Simulation Conference*, pages 1199 – 1208, Phoenix, AZ, December 1991.

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

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

[14] Paul A. Fishwick. Toward a Convergence of Systems and Software Engineering. *IEEE Transactions on Systems, Man and Cybernetics*, May 1996. Submitted for review.

[15] Paul A. Fishwick and Kangsun Lee. Two Methods for Exploiting Abstraction in Systems. *AI, Simulation and Planning in High Autonomous Systems*, pages 257–264, 1996.

[16] 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.

[17] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 1990. Second Edition.

[18] Ian Graham. *Object Oriented Methods.* Addison Wesley, 1991.

[19] Thomas G. Hallam, Ray R. Lassiter, Jia Li, and William KcKinney. Modeling Populations with Continuous Structured Models. In Donald L. DeAngelis and Louis J. Gross, editors, *Individual-Based Models and Approaches in Ecology*, pages 312–337. Chapman and Hall, New York, 1992.

[20] Donald Hearn and M. Pauline Baker. *Computer Graphics.* Prentice Hall, 1994.

[21] Nevin Heintze, Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. *The CLP($\mathcal{R}$) Programmer's Manual: Version 1.1*, November 1991.

[22] David R. C. Hill. *Object-Oriented Analysis and Simulation.* Addison-Wesley, 1996.

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

[24] Robert Kowalski. *Logic for Problem Solving.* Elsevier North Holland, 1979.

[25] Averill M. Law and David W. Kelton. *Simulation Modeling & Analysis.* McGraw-Hill, 1991. Second edition.

[26] Jin Joo Lee. *A Simulation-Based Approach for Decision Making and Route Planning.* PhD thesis, June 1996.

[27] Jin Joo Lee and Paul A. Fishwick. Real-Time Simulation-Based Planning for Computer Generated Force Simulation. *Simulation*, 63(5):299–315, November 1994.

[28] Kangsun Lee and Paul A. Fishwick. A Methodology for Dynamic Model Abstraction. *SCS Transactions on Simulation*, 1996. Submitted August 1996.

[29] William Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison Wesley, 1988.

[30] Aristid Lindenmeyer. Mathematical Models for Cellular Interaction in Development. *Journal of Theoretical Biology*, 18:280–315, 1968.

[31] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus.* Addison Wesley, 1989.

[32] Victor T. Miller and Paul A. Fishwick. Heterogeneous Hierarchical Models. In *Artificial Intelligence X: Knowledge Based Systems*, Orlando, FL, April 1992. SPIE.

[33] J. D. Murray. *Mathematical Biology.* Springer Verlag, 1990.

[34] Richard E. Nance. Simulation Programming Languages: An Abridged History. In *1995 Winter Simulation Conference*, pages 1307 – 1313, Washington, DC, December 1995.

[35] Donald A. Norman. *The Design of Everyday Things.* Currency Doubleday, New York, 1988.

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

[37] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

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

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

[40] G. Revesz. *Lambda Calculus Combinators and Functional Programming.* Cambridge University Press, 1988.

[41] Chell A. Roberts, Terrence Beaumariage, Charles Herring, and Jeffrey Wallace. *Object Oriented Simulation.* Society for Computer Simulation International, 1995.

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

[43] Ronald C. Rosenberg and Dean C. Karnopp. *Introduction to Physical System Dynamics.* McGraw-Hill, 1983.

[44] Jeff Rothenberg. Object-Oriented Simulation: Where do we go from here? Technical report, RAND Corporation, October 1989.

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

[46] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1990.

[47] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

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

[49] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 2 edition, 1991.

[50] Jean Thoma. *Bond Graphs: Introduction and Application.* Pergamon Press, 1975.

[51] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison Wesley, second edition, 1984.

[52] Wilfried F. Wolff. An Individual-Oriented Model of a Wading Bird Nesting Colony. *Ecological Modelling*, 72:75–114, 1994.

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

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

[55] Bernard P. Zeigler. *Theory of Modelling and Simulation*. John Wiley and Sons, 1976.

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

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

## 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 working at Newport News Shipbuilding and Dry Dock Co. (doing CAD/CAM parts definition research) and at NASA Langley Research Center (studying engineering data base models for structural engineering). His research interests are in computer simulation modeling and analysis methods for complex systems. He is a senior member of the IEEE and the Society for Computer Simulation. He is also a member of the IEEE Society for Systems, Man and Cybernetics, ACM and AAAI. Dr. Fishwick founded the `comp.simulation` Internet news group (Simulation Digest) in 1987. 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 `fishwick@cise.ufl.edu`.