IIE Transactions (1998) **30**, 811–820

A taxonomy for simulation modeling based on programming language principles

PAUL A. FISHWICK

Department of Computer and Information Science and Engineering, University of Florida, Building CSE, Room 301, Gainesville, FL 32611, USA. Email: fishwick@cise.ufl.edu

Received August 1995 and accepted June 1996

Although many diverse areas employ simulation models, no agreed-upon taxonomy has been developed to categorize and structure simulation models for all science and engineering disciplines. The discipline of simulation is often splintered due to this lack of structure, with *ad hoc* model classes such as 'discrete event,' 'continuous' and 'combined.' These classes most often reflect the method of execution used on a model rather than the design structure of the model. We present a uniform model design taxonomy whose categories are inspired from categories in programming language principles within the field of computer science. The taxonomy includes a set of primitive model types (conceptual, declarative, functional, constraint, spatial) and a way of integrating primitive model types together (multimodeling). These model types are discussed using a single application: a robot server for an assembly line. We have found this taxonomy enables simulators to more easily define and categorize their models as well as to understand how model types from seemingly disparate application areas are interrelated.

1. Introduction

Simulation is a tightly coupled and iterative three component process composed of: (1) *model design*; (2) *model execution*; and (3) *execution analysis* shown in Fig. 1. Model design can be accomplished without any *a priori* data or knowledge, or one may have sample data and prior knowledge about the normative model type used for the application. The word *model* is an overloaded term and can have many meanings depending on context. We proceed to first 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. In our methodology [1], a *model* is defined as a graph (a visual formalism consisting of nodes, arcs and labels) with two exceptions: a set of rules or equations. Computer code and programs are not considered to be models since code semantics are specified at too low a level. Likewise, formal methods are viewed to specify the formal semantics for models but do not focus on representing the kind of high-level form needed for modeling. We consider low-level constructs defining system dynamics to be *programs* or *formal specifications*, rather than *models*. Programs and formal specifications [2–4] are a vital ingredient in the simulation process since, without these methods, modeling approaches lack precision and cohesion. However, our definition of modeling is positioned at a different level: models can be translated into executable programs and formal specifications. Fishwick and Zeigler [5] demonstrated this translation using the DEVS [3] formalism for one particular type of visual multimodel (a finite state machine model controlling a set of constraint models). If one were to sketch a translation chain, it would be as follows: conceptual model $\Rightarrow$ executable model $\Rightarrow$ {Formalism,Program}. We focus specifically on modeling as the first two links in this chain. An in-depth discussion of different model types including conceptual, constraint and multimodel is presented in subsequent sections. Models are the visual high-level constructs that we use to communicate system dynamics without the need for frequent communication of low-level formalism and semantics.

We will first motivate the need for a new taxonomy by stressing issues and problems with the existing taxonomies for computer simulation modeling. We then specify our modeling taxonomy, and illustrate our model types using the scenario of the single server queue. This type of queue is ubiquitous, simple and yet provides a way of describing the new taxonomy. The paper is organized as follows: we present the new modeling taxonomy. Then we describe the example scenario that is used to illustrate all model types: conceptual, declarative, functional, constraint, spatial and multimodel. We close with a summary of the taxonomy and its advantages, with future goals to be achieved.

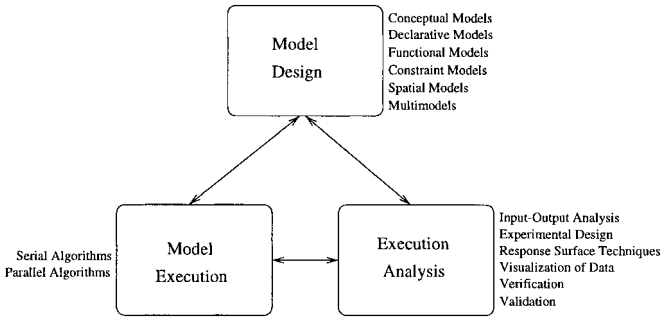

Fig. 1. The study of computer simulation: three subfields.

2. Motivation

Given that various taxonomies exist for modeling, it may not be clear why we create yet another taxonomy. Here are our key reasons:

1. *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 which 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;
2. *Object-Oriented Design:* the new taxonomy is one based on object-oriented (OO) design methodology since it is developed as an extension to OO design. Existing OO design for software engineering does not usually include the concept of modeling. Our design extends the design approaches in software engineering to employ both static and dynamic models for physical objects;
3. *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;
4. *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;
5. *Models versus Programs:* we draw a clear parallel between the new taxonomy 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 category types.

While there has been significant coverage in the simulation literature for analysis methods [6,7], the general area of modeling for simulation has lacked uniformity and indepth 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 [6,7] 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 [8] has a particular form regardless of the way in which it is executed. Ideally, then, we would like to create a model category which 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 [9] or animation scripts [1]) or whether a GPSS or Simscript program [10] 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 separated into different categories using the existing 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 containing directed arcs. Likewise, this flow is directly analogous to functional composition in functional programming languages.

The field of computer science contains categories of programming languages [11,12] that serve to create three of the following six model categories: conceptual, declarative, functional, constraint, spatial and multimodel.

Declarative programming semantics are those where data and variable declarations constitute the expression of the semantics; one specifies how variables change in the program by declaring present and future variable values. The form of declarative models is either graphical (automata,chains) or textual (patterns,logical inferences). Prolog [13] is an example of a declarative language with a textual orientation. Functional languages, such as Lisp [14], specify how variables change using compositional function mappings. These mappings conform exactly to the mathematical definition of the term 'function.' Constraint programming languages [15] allow both forward and backward chains to be created to 'solve' for variable values. These same categories can be used to organize dynamic *models* just as they currently do for *programs*, with modeling and programming having similar roots [16].

In Table 1, we show the base categories along with application areas and some sample model types used by those areas. Multimodels are not listed since they are models containing base category models as components.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 [17]) 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 [18,19], are very similar in principle. If programming language statements are not viewed declaratively, they usually are defined using functional semantics. The languages Lisp [14] and ML [20] 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 [15,21] reflect a way of programming where procedures and declarations are insufficient. The constraint language CLP($\mathscr{R}$) [22] (Constraint Logic Programming) represents this type of language. Also, the next generation of Prolog (Prolog III) is constraint oriented. In constraint models, the focus is on a model structure, which involves basic balances of units such as mass and energy.

The remaining three types of models (conceptual, spatial and multimodel) have the following justifications. Conceptual models are highlighted in Artificial Intelligence (AI). In AI, we find a plethora of potential conceptual model types including semantic networks, logic-based formalisms (without a time base), frames and other schemata. We have found the object oriented (OO) paradigm [23–25] to encapsulate a favorable way of encoding conceptual models. One should begin to define a system by first defining attributes, methods and classes, along with aggregation and association hierarchies. An aggregation hierarchy is one involving a 'part of' relation between the children and the parent in a tree. A generalization relation involves a 'is a kind of' relation. The resulting hierarchies are not executable but they serve as a mechanism to organize objects and their relations to one

Table 1. Sample model categories and associated applications

Base model type	*Application area*	*Modeling technique*
Conceptual	Artificial intelligence	Semantic networks, frames
	Software engineering	Object oriented designs
	Soft systems science	Primitive model types
Declarative	Computer science (theory)	Automata
	Computer animation	Scripts and tracks
	Engineering	Task checklists
	Programming languages	Logic programs
	Operating systems	Petri nets
	Physics	Space–time diagrams
Functional	Automatic control	Block models
	Electrical engineering	Digital circuits
	Performance analysis	Queuing networks
	Biology and medicine	Compartmental models
	Industrial systems	System dynamics
Constraint	Dynamics	Equations
	Electrical engineering	Analog circuits
	Mechanical engineering	Bond graphs
Spatial	Complex systems	Cellular automata
	Engineering	Finite element modeling, PDEs

another. Spatial models, such as partial differential equations (PDEs) and cellular automata, are common where we have detailed parametric information about the system from a geometric perspective. We organize spatial models using class and object aggregation, which are approaches in object-oriented design. Multimodels are defined as models containing other models. This recursive definition allows for powerful model-building to take place where large scale models can be constructed out of simpler primitive ones.

3. Modeling taxonomy

Figure 2 illustrates the modeling taxonomy. The top level of Fig. 2 refers to the multimodel type since this type is composed of all sub-types previously discussed: 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:

- *Conceptual* models are either in pictorial form or in natural language 'text' form. For our purposes, we have chosen to represent conceptual models as aggregation and generalization hierarchies, commonly found in object-oriented design [24];
- *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 [26];
- *Spatial* models have a focus on either the whole space or an entity. Space-based orientations involve the updated state of the *entire space* by convolution of a template over that space (as with the numerical solution of PDEs). Entity-based orientations involve the dynamics of individual objects (i.e., entities) which move around a space.

4. Scenario

Having provided justification for the new model taxonomy, we now proceed to illustrate the different dynamic model types using a single server queue. The single server queue is ubiquitous in engineering and business. In science, it is present where higher order living entities must block each other, waiting for service. For lower order entities and non-organic materials, the general concept of *capacitance* serves to emulate queuing to some extent.

Consider a two-link robot arm that accepts parts, performs a service, and releases the parts. The overall top view of the process is shown in Fig. 3. Figure 4 displays a subset of the robot tasks that are performed:

1. *Grasp part*: move from the default reset position to grab the next part waiting for service;
2. *Move part to camera*: move the part to the camera so that it can be inspected for flaws;
3. *Inspection*: execute image processing and vision programs to segment the image and determine if a correct match can be made against a stored 'template' image;
4. *Move part to release point*: if the part is acceptable, it is moved back to the conveyor at the release point. Otherwise, the part is placed in a bin for rejected parts;
5. *Release part*: release the part onto the conveyor;
6. *Reset*: reset the robot so that it is prepared for the next inspection. Synchronize the motors and end-effectors so that they are correctly positioned.

The incoming conveyor for the robot stops when the infrared transmitter-receiver pair indicates a part is ready.

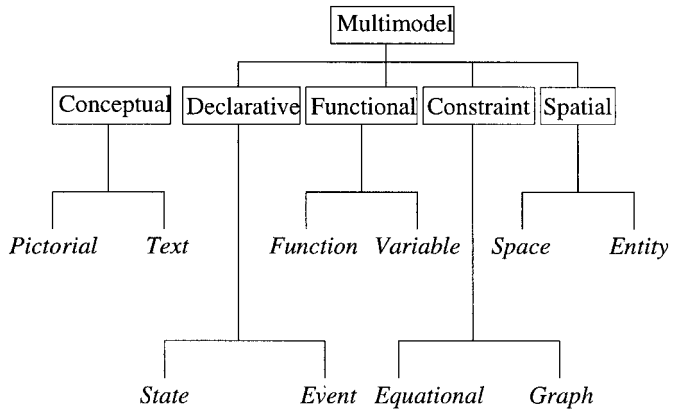

Fig. 2. Model taxonomy.

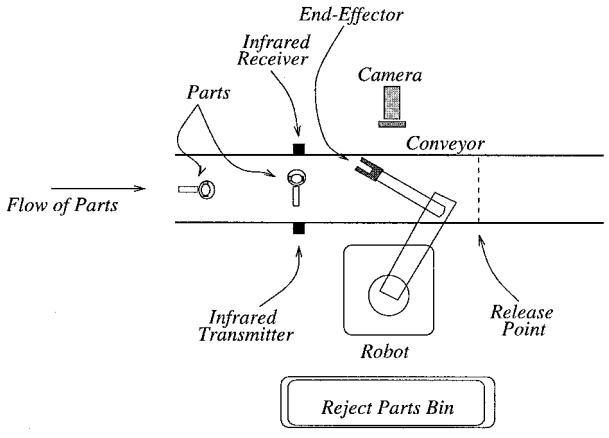

Fig. 3. Top view of single server queue process.

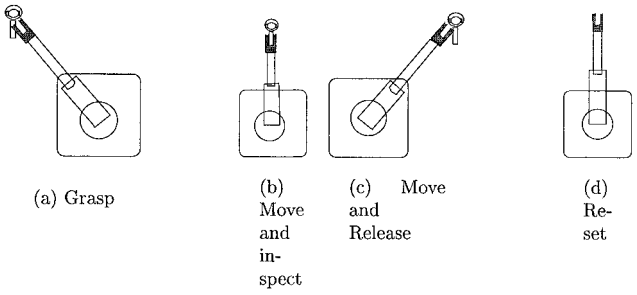

Fig. 4. Phases of robot server *pick* and *place* operations.

The part is then grabbed by the robot's end-effector and presented to the camera for visual inspection. After the part is grasped and moved, the conveyor resumes operation to deliver the next part for service. Service of a part is based on the speed of the arm as well as the image processing time. Let us now proceed with defining each model for this scenario.

5. Conceptual model

Conceptual models are 'first cut' descriptions of what we know about a process. We could, for instance, describe our knowledge of the robot and the process using a variety of schemata including frames, semantic networks or entity-relationship diagrams. Instead, we choose to follow the object-oriented paradigm, which suggests that we form classes along with encapsulated attributes and methods. Specifically, we form a hierarchy or set of hierarchies which specify two hierarchical relations among all classes: aggregation and generalization. Figure 5 provides a conceptual model which contains two aggregation relations (using a box to designate the relation) and one generalization relation (using a circle). In this figure, we have decided to include both types of relations in the same hierarchy, but it is acceptable to separate

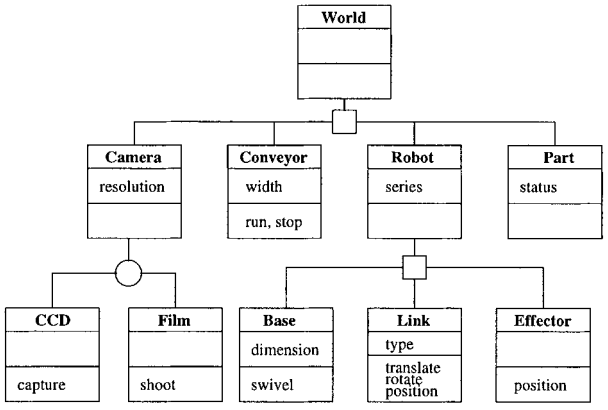

Fig. 5. Conceptual model including aggregation and generalization.

hierarchies if needed. Other relations, aside from generalization and aggregation, are possible; however, we focus on these two relations since they involve an implicit transfer of class structure. For generalization, this transference is called *inheritance*—the inheriting of attributes and methods from a base class (such as **Camera**) to derived classes (child nodes of **Camera: CCD** and **Film**). For aggregation, the transfer of class structure is bottom-up instead of top-down as it was for generalization. A **Robot** class logically aggregates all sub-class structure. The topmost class **World** contains the objects below it. Therefore the four objects **Camera, Conveyor, Robot** and **Part** aggregate together to form **World**. There are two types of cameras: **CCD** and **Film**. As it turns out, we will not be using any camera attributes or methods in our models, but we include this knowledge in our conceptual model anyway. The larger goal of conceptual modeling is to capture all knowledge about a physical set of objects, whereas a single modeling exercise may use only a subset of what is stored in the conceptual model.

There are several reasons for using an object-oriented design for the conceptual model, including the use of advantageous features such as polymorphism and inheritance [27]; however, the primary reason is that the design is 'close' to reality—at least in the way that we categorize the world with natural language. Conceptual models comprise both aggregation and generalization hierarchies [23,24]. Generalization hierarchies specify relationships based on type: a robot may be of several types such as welding, pick-and-place or inspection, or some combination of these. Aggregation hierarchies specify a relation based on composition: a robot is composed of a base, links and an end-effector. Within an object (ref. Fig. 5) attributes are above the horizontal center line and methods are below the line. The conveyor's attribute of *width* reflects the width of the conveyor belt whereas the method *run* describes a method that is associated with the belt drive. An example of polymorphism is shown in the link and effector objects. The method *position* is common to both. The program knows which method to use only through determining which object is affected at run time.

Once the conceptual model has been constructed, one must 'fill in' the attributes and methods for each class. Some attributes and methods are shown in Fig. 5. We extend the existing OO design paradigm to include modeling in the following sense. An attribute can be 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++ [27]. We define a static model as a graph of objects and a dynamic model as a graph of attributes and methods. The model types we focus on in this paper are all dynamic, however, the concept of static model complements the concept of dynamic model: methods operate

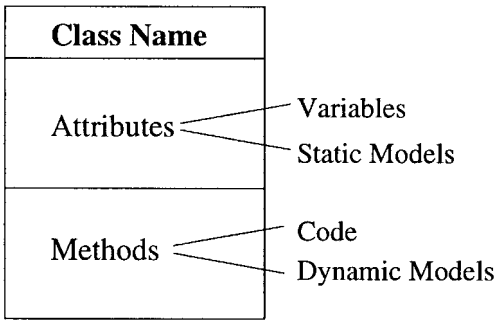

Fig. 6. Structure of a class.

on attributes to effect change in an object. Likewise, dynamic models operate on static models to effect change. Objects are instantiations of classes. For example, one creates a robot object called *Robot*35 by making an instance of the class **Robot**.

6. Declarative model

Models that permit dynamics to be encoded as state-to-state or event-to-event transitions are *declarative*. The idea behind declarative modeling is to focus on the structure of the state (or event) from one time period to the next, while de-emphasizing functions or constraints that define the transition. Models such as finite state automata [28], Markov models, event graphs [9] and temporal logic models [29] fall into the declarative category. Declarative models are state-based (FSAs), event-based (event graphs) or hybrid (Petri nets [8]). For the robot scenario, example states for the robot object are 'grasping,' 'inspecting' (for high abstraction levels) or the number of parts waiting for service on the conveyor (for low abstraction levels). The level of information and abstraction of models is discussed more in the *Multimodel* section. The following types of declarative models represent an appropriate subset of declarative modeling types:

- *Stochastic process*: a Markov model can be used to specify state-to-state transitions based on arrival λ and departure μ rates;
- *Event graph*: two events (arrival,departure) are used;
- *Petri net*: the part waiting at the infrared beam (in Fig. 3) can be expressed as a resource contention where the part contends for the robot *resource*.

Figure 7 displays these three model types. The first type (Markov) represents a stochastic process that has a memory of unit length — the state of the system at the current time is dependent on the state at the previous time, and not on any times prior to this. The declarative nature of the model shows itself because our model is composed, literally, of state specifications. The event

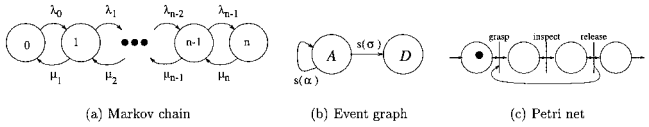

Fig. 7. Declarative model types.

graph in Fig. 7 is somewhat different because it illustrates a focus on events that transpire, A (arrival) and D (departure). Events A and D are not primitive since these events are actually place holders (or sets) including primitive events, each of which represents an event for one specific part. This type of modeling reflects the capability of declarative models to use patterns as well as unit state/event descriptors. Event A schedules another arrival using an exponential distribution for interarrival times α (i.e., a Poisson input process), and A also schedules D to occur in σ (service) time. The term $s(X)$ means 'schedules using parameter X.' The Petri net in Fig. 7 models the robot server by stressing resource contention as a key model design aspect. The part (represented by a solid black circle) flows through a simplified assembly-line system, while encountering three types of transactions: grasp by the robot, inspection, and release of the part. The key aspect of the Petri net is seen in the feedback arc, which dictates that the *next* part on the line must wait for the robot to come back to grasp it before it can proceed to be serviced.

7. Functional model

Functional models represent a directional flow of a signal (discrete or continuous) among transfer functions (boxes or blocks). When the system is viewed as a set of blocks communicating with messages or signals, the functional paradigm takes hold. The use of functional models is found in control engineering [30,31] (continuous and discrete time signals) as well as in queuing networks (discrete signals). Some functional systems focus not so much on the functions, but more on the variables. Such models include signal flow graphs, compartmental models [32], and Systems Dynamics graphs [33]. In this latter group of models, the function is often an implicit linear one, with the pronounced variables shown in the graph representing state variables along with parameters.

The functional way of modeling the robot is to consider the robot as a transfer function, modeling a relation between input and output. The input signal, for our purpose, is a continuous-time, discrete-state signal reflecting the arrival of parts. The output signal specifies the time-based departure of parts. Consider Table 2, which provides some data for the parts (P), their time since last arrival ($TSLA$), the arrival clock time (ACT), service time (ST), service begin time (SB), service end time (SE), part wait time (PW) and robot server idle time (SI). Figure 8

Table 2. Times for four successive parts

P#	*TSLA*	*ACT*	*ST*	*SB*	*SE*	*PW*	*SI*
1	5.0	5.0	6.5	5.0	11.5	0.0	5.0
2	6.3	11.3	7.3	11.5	18.8	0.2	0.0
3	6.0	17.3	5.2	18.8	24.0	1.5	0.0
4	7.8	25.1	6.2	25.1	31.3	0.0	1.1
Totals						1.7	6.1

displays the functional model for the robot as a transfer function, along with the input and output trajectories. The model, in this case, is the block in the center; the trajectories are shown to illustrate the response. For a single server this model seems fairly pointless; however, for multiple servers coupled in a network, the functional model represents an effective modeling approach. We still have not described, in modeling nomenclature, how the transfer function operates. It is often useful to put one type of model in another for this purpose. For the functional model, we can refine the block into a declarative model to create a multimodel composed of two models.

8. Constraint model

There are two types of constraint models: *equational* and *graph-based*. Constraint models are models where a balance (or constraint) is at the heart of the model design. In such a case, an equation is often the best characterization of the model since a directional approach such as functional modeling is insufficient. Equational systems include difference models, ordinary differential equations and delay differential equations. Graphical models such as bond graphs [26,34] and electrical network graphs [35] are also constraint based. To model the robot server by constraint modeling techniques, we encode the model as a balance of parts in Equation 1:

$$\textit{Transition Probability Rate} = \textit{Parts Arriving} - \textit{Parts Released} \tag{1}$$

This is represented by Kolmogorov's forward equation for a *birth-death* process in Equation 2:

$$\underbrace{\frac{\mathrm{d}}{\mathrm{d}t}P_{ij}(t)}_{\text{Rate}} = \underbrace{\lambda_{j-1}P_{i,j-1}(t) + \mu_{j+1}P_{i,j+1(t)}}_{\text{Parts Arriving}} - \underbrace{(\lambda_j + \mu_j)P_{ij}(t)}_{\text{Parts Released}} \tag{2}$$

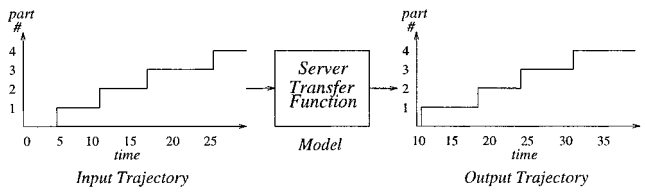

Fig. 8. Functional model for the server.

9. Spatial model

If a system is spatially decomposed as for cellular automata (CA) [36,37], Ising systems, PDE-based solutions or finite element models, then the system can be modeled using a spatial modeling technique. Spatial models are used to model systems in great detail, where individual pieces of physical phenomena are modeled by discretizing the geometry of the system. Spatial models are 'entity-based' or 'space-based.' Entity-based spatial models focus on a fixed space where the entity dynamics are given whereas space-based models focus on how the space changes by convolving a template over the space at each time step. PDEs are space-based where the template defines the integration method. L-Systems [38] are entity-based since the dynamics are based on how the organism grows over a fixed space. A CA represents a simple way of modeling a single server queue. Figure 9 illustrates a CA with parts waiting for service from the robot. The CA is one dimensional and is represented as a contiguous group of 8 squares. Each square can be blank or contain one of the icons in Fig. 10. Since the CA represents a discrete time approximation, we use integer values for times. The service time is 4 units and is represented by a cell that changes state for the correct number of phases before accepting the next part to be serviced. The solid circle represents a part. The shaded circle and polygons represent the phases of the server and the cross is a *wildcard* that matches any other cell. The rules in Fig. 10 are 'block rules' [36] and are used as patterns used in a convolution procedure over the background CA space. A rule is applied by a search for the pattern on the left hand side of the rules in Fig. 10. If there is a match of the pattern against any part of the one dimensional space, the original part of that space (against which the pattern matched successfully) is replaced by the right hand part of the rule. For example, consider the CA at time *I* in Fig. 9.

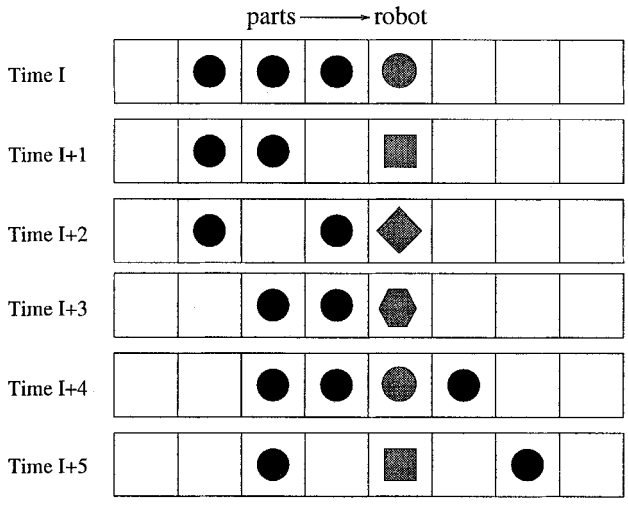

Fig. 9. Cellular automaton for single server queue.

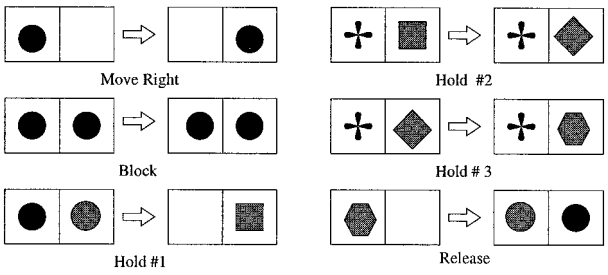

Fig. 10. CA rules.

We ask ourselves: which left hand block in Fig. 10 matches some portion of the CA in Fig. 9 at time *I*? The *Block* rule in Fig. 10 matches, but this results in no change, whereas the *Hold#1* rule matches and results in the replacement of the matching part of Fig. 9 at time *I* with the corresponding right hand block in Fig. 10 for *Hold#1*, resulting in a new CA for time $I + 1$ in Fig. 9. The simulation of the CA model proceeds using this same logic. There is a lag in the queue as parts move from left to right toward service. This is an artifact of the CA since, with a continuous belt feed, this lag would not appear—the queue elements would move in unison. Another set of CA rules can associate a number of contiguous cells for use by the server. This represents a transport delay and would achieve the same effect as using the multiple state cells in Fig. 10.

More complex spatial models can also represent the single server queue. Most 2D and 3D graphically oriented simulations provide for queuing as a by-product of their visual layout. The simulation of a server and queue involves two important requirements: (1) to store cells or parts; and (2) to detect collisions or adjacent, occupied spatial areas. The first requirement, which is normally satisfied in non-spatial models through the use of facility queues, is satisfied automatically by employing spatial data structures [39,40]. The second requirement is satisfied through adjacency tests or collision detection algorithms.

10. Multimodel

Large scale models are built from one or more abstraction levels, each level being designed using one of the aforementioned primitive model types. The lowest level of abstraction for a system will often implement a spatial model whereas the highest level may use a declarative finite state machine. Intermediate levels will often use functional and constraint techniques. Models that are composed of other models are termed *multimodels* [5,41,42]. By utilizing abstraction levels, we can switch levels during the simulation and use the abstraction most appropriate at that given time. This approach gives us multiple levels of explanation and is computationally more efficient than simulating the system at one level. Multimodels will be necessary as we strive for a more *holistic* (bringing together formerly disparate model types) attitude toward modeling.

We will discuss two aspects of multimodeling for the robot scenario: Aggregation (Abstraction) and Disaggregation (Refinement). Even though we are modeling a single server queue, we could easily imagine modeling an entire assembly line that includes our robot server as one component. In doing this, we create higher level transfer functions that serve as aggregates for the lower level servers. This is an example of aggregation–lumping together low level system components to create a simplified, aggregate component. An example of disaggregation (or process refinement) is a sub-model that models in greater detail the semantics of *service*. In our previous models, we have 'abstracted away' the issues of automatic control and physical variables such as force, torque and position; however, many modelers may be interested in observing the system at this level of detail. Let us analyze the procedure of *service*:

1. Part is sensed by the infrared beam;
2. Robot is initially in a reset state;
3. Gripper moves to part that is waiting at the acquisition point;
4. Gripper opens and closes on part;
5. Part is moved to camera for inspection;
6. Part is either moved to the 'rejects bin' or moved to the release point;
7. Robot is reset.

If the control for the robot is fully automatic, there needs to be a control program set up to guide the robot through these actions. Two types of control are important for our purpose: *position* and *speed* control. The control system should produce a stable movement so that the gripper moves more or less linearly with input. Figure 11 demonstrates one of our sub-models necessary for the correct

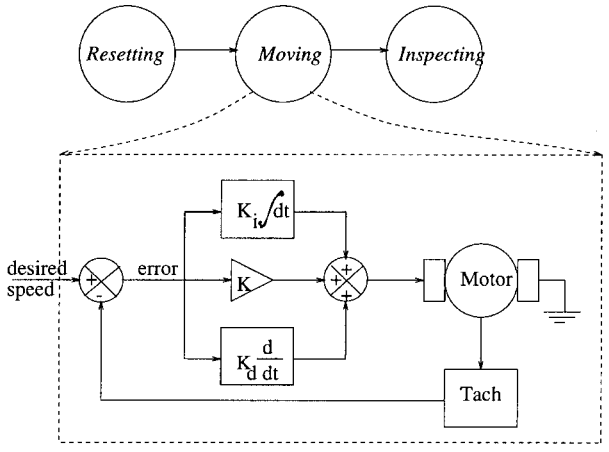

Fig. 11. PID speed controller for robot arm.

operation of the robot: a proportional-integral-derivative (PID) controller. This represents one of several possible methods for compensation. This controller controls the speed of the robot.The desired speed is issued as a signal to the first comparator (circle with a cross-bar) and is balanced by the actual speed reading from the tachometer (Tach) which senses the motor (actuator) speed. An error is produced as a result, and this error is fed into the PID section. This model is a multimodel because there are two model types present: (1) the declarative automaton model of low detail; and (2) the functional control model of high detail. A section of the high level model (finite state automaton) has three states (phases) in which the robot finds itself during its operations: (1) resetting; (2) moving and (3) inspecting. The 'Moving' state is refined (or disaggregated) to include the PID controller. Multimodels with more than two levels and model types are detailed in Fishwick [1].

11. Summary

The existing taxonomy for simulation modeling was found to have several flaws, which are rectified using the new taxonomy. Some flaws include the lack of a clear distinction between syntax and semantics, and an unnatural separation of model types whose form is identical (as for functional block networks containing server functions versus functional block networks containing transfer functions). The presented taxonomy of base model types, including multimodeling, provides a cohesive view on system modeling for simulation with an eye to clustering types together based on programming language categories. Programs and models are similar in that both have syntax (form) and semantics (execution). The similarity between models and programs lends support to providing a taxonomy for modeling which is isomorphic to the taxonomy for programming language principles.

We have used this taxonomy for the past four years at the University of Florida in both undergraduate and graduate classes in computer simulation. These classes were taught within a Computer Science curriculum, therefore, students were able to easily identify with terms such as 'declarative,' 'functional' and 'constraint' since these terms are associated with programming language principles. Students with an artificial intelligence background could easily identify with conceptual (symbolic) modeling, and engineering students were well versed in dynamics and the need for equationally based conservation laws and spatial models. Our emphasis is for students to first create conceptual models for a physical scenario, by defining the aggregation and generalization class hierarchies. After this procedure, one fills in the attributes and methods, some of which represent static and dynamic models. At this stage, the physical scenario is well-organized in terms of this extended object-oriented design. Simulation is made more comprehensible.

Acknowledgments

I would like to acknowledge the following funding sources which have contributed towards our study of modeling and implementation of a multimodeling simulation environment: (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 (3) the 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] Fishwick, P.A. (1995) *Simulation Model Design and Execution: Building Digital Worlds*, Prentice Hall, Englewood Cliffs, NJ.
[2] Zeigler, B.P. (1972) Towards a formal theory of modelling and simulation: structure preserving morphisms. *Journal of the Association for Computing Machinery*, **19**(4), 742–764.
[3] Zeigler, B.P. (1989) DEVS representation of dynamical systems: event-based intelligent control. *Proceedings of the IEEE*, **77**(1), 72–80.
[4] Praehofer, H. (1991) Systems theoretic formalisms for combined discrete-continuous system simulation. *International Journal of General Systems*, **19**(3), 219–240.
[5] Fishwick, P.A. and Zeigler, B.P. (1992) A multimodel methodology for qualitative model engineering. *ACM Transactions on Modeling and Computer Simulation*, **2**(1), 52–81
[6] Banks, J., Carson, J. S. and Nelson, B. L. (1996) *Discrete Event System Simulation*, 2nd edn, Prentice Hall, Englewood Cliffs, NJ.
[7] Law, A.M. and Kelton, D.W. (1991) *Simulation Modeling and Analysis*, 2nd edn, McGraw-Hill, New York, NY.
[8] Peterson, J.L. (1981) *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ.
[9] Schruben, L. (1995) *Graphical Simulation and Modeling using SIGMA for Windows*, 3rd edn, The Scientific Press Series, Boyd and Fraser Publ. Co., Danvers, MA.
[10] Nance, R.E. (1995) Simulation programming languages: an abridged history, in *Proceedings of the 1995 Winter Simulation Conference*, pp. 1307–1313, Society for Computer Simulation International, San Diego, CA.
[11] Ghezzi, C. and Jazayeri, M. (1982) *Programming Language Concepts*, John Wiley, New York, NY.
[12] Meyer, B. (1990) *Introduction to the Theory of Programming Languages*, Prentice Hall, Englewood Cliffs, NJ.
[13] Colmerauer, A. (1985) Prolog in 10 figures. *Communications of the ACM*, **28**(12), 1296–1310.
[14] Winston, P.H. and Horn, B.K.P. (1984) *LISP*, 2nd edn, Addison Wesley, Reading, MA.
[15] Leler, W. (1988) *Constraint Programming Languages: Their Specification and Generation*, Addison Wesley, Reading, MA.
[16] Fishwick, P.A. (1996) Toward a convergence of systems and software engineering. *IEEE Transactions on Systems, Man and Cybernetics*, Technical Report TR96-005, University of Florida (`http://www.cise.ufl.edu/~fishwick/tr/tr96-005.html`).

[17] Kowalski, R. (1979) *Logic for Problem Solving*, Elsevier North Holland, Amsterdam.
[18] Michaelson, G. (1989) *An Introduction to Functional Programming through Lambda Calculus*, Addison Wesley, Reading, MA.
[19] Revesz, G. (1988) *Lambda Calculus Combinators and Functional Programming*, Cambridge University Press, Cambridge, UK.
[20] Paulson, L.C. (1991) *ML for the Working Programmer*, Cambridge University Press. Cambridge UK.
[21] Borning, A.H. (1979) THINGLAB – a constraint-oriented simulation laboratory. Technical report, Xerox PARC, Palo Alto, CA.
[22] Heintze, N., Jaffar, J., Michaylov, S., Stuckey, P. and Yap, R. (1991) *The CLP (ℛ) Programmer's Manual: Version 1.1* Technical Report, Department of Computer Science, University of Melbourne, Australia.
[23] Booch, G. (1991) *Object Oriented Design*, Benjamin Cummings, Reading, MA.
[24] Rumbaugh, J., Blaha, M., Premerlani, W., Frederick, E. and Lorenson, W. (1991) *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ.
[25] Zeigler, B.P. (1990) *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, Boston, MA.
[26] Breedveld, P.C. (1986) A systematic method to derive bond graph models, in *Proceedings of the Second European Simulation Congress*, Antwerp, Belgium, Gordon and Breach Publishers, Amsterdam.
[27] Stroustrup, B. (1991) *The C++ Programming Language*, 2nd edn. Addison Wesley, Reading, MA.
[28] Hopcroft, J.E. and Ullman, J.D. (1979) *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, Reading, MA.
[29] Moszkowski B. (1986) *Executing Temporal Logic Programs*, Cambridge Press, Cambridge, UK
[30] Dorf, R.C. (1986) *Modern Control Systems*, Addison Wesley, Reading, MA.
[31] Ogata, K. (1970) *Modern Control Engineering*, Prentice Hall, Englewood Cliffs, NJ.
[32] Jacquez, J.A. (1985) *Compartmental Analysis in Biology and Medicine*, 2nd edn, University of Michigan Press, Ann Arbor, MI.
[33] Roberts, N., Andersen, D., Deal, R., Garet, M. and Shaffer, W. (1983) *Introduction to Computer Simulation: A Systems Dynamics Approach*, Addison-Wesley, Reading, MA.
[34] Karnopp, D.C., Margolis, D.L. and Rosenberg, R.C. (1990) *System Dynamics*, John Wiley and Sons, New York, NY.
[35] Raghuram, R. (1989) *Computer Simulation of Electronic Circuits*, John Wiley, New York, NY.
[36] Toffoli T. and Margolus, N. (1987) *Cellular Automata Machines: A New Environment for Modeling*, 2nd edn, MIT Press, Cambridge, MA.
[37] Wolfram, S. (1986) *Theory and Applications of Cellular Automata*, World Scientific Publishing, Singapore, (includes selected papers from 1983–1986).
[38] Prusinkiewicz, P. and Lindenmeyer, A. (1990) *The Algorithmic Beauty of Plants*, Springer-Verlag, New York, NY.
[39] Samet, H. (1990) *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA.
[40] Samet, H. (1990) *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA.
[41] Fishwick, P.A. (1992) An integrated approach to system modelling using a synthesis of artificial intelligence, software engineering and simulation methodologies. *ACM Transactions on Modeling and Computer Simulation*, **2**(4), 307–330.
[42] Fishwick, P.A. (1993) A simulation environment for multimodeling. *Discrete Event Dynamic Systems: Theory and Applications*, **3**, 151–171.

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 Fellow of the Society for Computer Simulation and a senior member of the IEEE. 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*.

Contributed by the Simulation Department