Theory of Modeling and Simulation

Integrating Discrete Event and Continuous Complex Dynamic Systems

Second Edition

BERNARD P. ZEIGLER
Electrical and Computer Engineering Department
University of Arizona
Tucson, Arizona

HERBERT PRAEHOFER
Institute of Systems Science
Johannes Kepler University
Linz, Austria

TAG GON KIM
Department of Electrical Engineering
Korea Advanced Institute of Science and Technology
Taejon, Korea

ACADEMIC PRESS

An Imprint of Elsevier

Amsterdam Boston Heidelberg London New York Oxford
Paris San Diego San Francisco Singapore Sydney Tokyo

3

Modeling Formalisms and Their Simulators

3.1 Introduction

This chapter presents, in an informal manner, the basic modeling formalisms for discrete time, continuous, and discrete event systems. It is intended to provide a taste of each of the major types of models, how we express behavior in them, and what kinds of behavior we can expect to see. Each modeling approach is also accompanied by its prototypical simulation algorithms. The presentation employs commonly accepted ways of presenting the modeling formalisms. It does not presume any knowledge of formal systems theory and therefore serves as an independent basis for understanding the basic modeling formalisms that will be cast as basic system specifications (DESS, DTSS, and DEVS) after the system theory foundation has been laid.

3.2 Discrete Time Models and Their Simulators

Discrete time models are usually the most intuitive to grasp of all forms of dynamic models. As illustrated in Fig. 1, this formalism assumes a stepwise mode of execution. At a particular time instant the model is in a particular state and it defines how this state changes—what the state at the next time instant will be. The next state usually depends on the current state and also what the environment's influences currently are.

Discrete time systems have numerous applications. The most popular are in digital systems where the clock defines the discrete time steps. But discrete time systems are also frequently used as approximations of continuous systems. Here a time unit is chosen, e.g., 1 second, 1 minute, or 1 year, to define an artificial clock, and the system is represented as the state changes from one "observation" instant to the next. Therefore, to build a discrete time model, we have to define how the current state and the input from the environment determine the next state of the model.

The simplest way to define the way states change in a model is to provide a table such as Table 1. Here we assume there are a finite number of states and inputs. We write down all combinations of states and inputs and next states and outputs for each. For example, let the first column stand for the current state of the model and the second column for the input it is currently receiving. The table gives the next state of the model in column 3 and the output it produces in column 4. In Table 1, we have two states (0 and 1) and two inputs (also 0 and 1). There are four combinations, and each one has an asso-

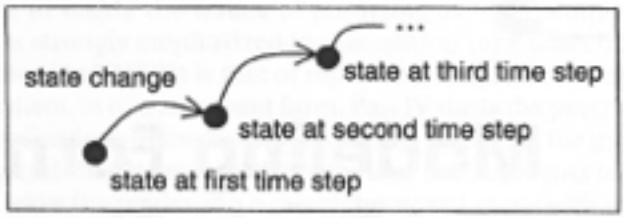

Figure 1 Stepwise execution of discrete time systems.

ciated state and output. The first row says that if the current state is 0 and the input is 0, then the next state will be 0 and the output will be 0. The other three rows give similar information for the remaining state/input combinations.

In discrete time models, time advances in discrete steps, which we assume are integer multiples of some basic period such as 1 second, 1 day or 1 year. The transition/output table just discussed would then be interpreted as specifying state changes over time, as in the following:

> If the state at time t is q and the input time t is x, then the state at time $t+1$ will be $\delta(q,x)$ and the output y at time t will be $\lambda(q,x)$.

Here δ is called the *state transition* function and is the more abstract concept for the first three columns of the table. λ is called the *output function* and corresponds to the first two and last columns. The more abstract forms, δ and λ, constitute a more general way of giving the transition and output information. For example, Table 1 can be summarized more compactly as

$$\delta(q,x) = x \text{ and}$$
$$\lambda(q,x) = x,$$

which say that the next state and current output are both given by the current input. The functions δ and λ are also much more general than the table.

Table 1 Transition/Output Table for a Delay System

Current state	Current input	Next state	Current output
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	1

They can be thought about and described even when it is tedious to write a table for all combinations of states or inputs—or indeed, when they are not finite and writing a table is not possible at all.

A sequence of states, $q(0), q(1), q(2), \ldots$ is called a *state trajectory*. Having an arbitrary initial state $q(0)$, subsequent states in the sequence are determined by

$$q(t+1) = \delta(q(t), x(t)).$$

Similarly, a corresponding *output trajectory* is given by

$$y(t) = \lambda(q(t), x(t)).$$

Table 2 illustrates state and output trajectories (third and fourth rows, respectively) that are determined by the input trajectory in the second row.

3.2.1 Discrete Time Simulation

We can write a little algorithm to compute the state and output trajectories of a discrete time model given its input trajectory and its initial state. Such an algorithm is an example of a simulator (as defined in Chapter 2).

Note that the input data for the algorithm corresponds to the entries in Table 3.

```
T_i = 0, T_f = 9 the starting and ending times, here 0 and 9
x(0) = 1, . . . , x(9) = 0 the input trajectory
q(0) = 0 the initial state

t = T_i
while (t <= T_f) {
  y(t) = λ(q(t), x(t))
  q(t+1) = δ(q(t), x(t))
}
```

Executing the algorithm fills in the blanks in Table 3 (except for those checked).

Exercise: Execute the algorithm by hand to fill in Table 3. Why are the marked squares not filled in?◆

Table 2 State and Output Trajectories

Time	0	1	2	3	4	5	6	7	8	9	
Input trajectory	1	0	1	0	1	0	1	0	1	0	
State trajectory	0	1	0	1	0	1	0	1	0	1	0
Output trajectory	1	0	1	0	1	0	1	0	1	0	

Table 3 Computing State and Output Trajectories

Time	0	1	2	3	4	5	6	7	8	9	
Input trajectory	1	0	1	0	1	0	1	0	1	0	√
State trajectory	0										
Output trajectory											√

Exercise: Table 4 is for a model called a binary counter. Hand simulate the model in the table for various input sequences and initial states. Explain why it is called a binary counter.◆

3.2.2 Cellular Automata

Although the algorithm just discussed seems very simple indeed, the abstract nature of the transition and output functions hides a wealth of potentially interesting complexities. For example, what if we connected together systems (as in Chapter 1) in a row with each system connected to its left and right neighbors, as shown in Fig. 2?

Imagine that each system has two states and gets the states of its neighbors as inputs. Then there are eight combinations of states and inputs as listed in Table 5. We have left the next state column blank. Each complete assignment of a 0 or 1 to the eight rows results in a new transition function. There are $2^8 = 256$ such functions. Suppose we chose any one such function, started

Table 4 State and Output Trajectories

Current state	Current input	Next state	Current output
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

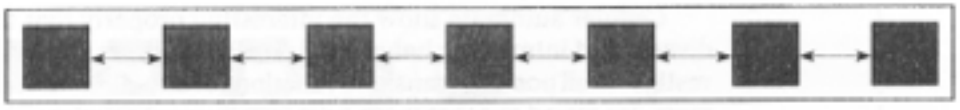

Figure 2 One-dimensional cell space.

each of the components in Fig. 2 in one of its states, and applied an appropriate version of the preceding simulation algorithm. What would we observe?

The answer to such questions is the concern of the field of *cellular automata*. A cellular automaton is an idealization of a physical phenomenon in which space and time are discretized and the state sets are discrete and finite. Cellular automata have components, called *cells*, which are all identical with identical computational apparatus. They are geometrically located on a one-, two-, or multidimensional grid and connected in a uniform way. The cells influencing a particular cell, called the *neighborhood* of the cell, are often chosen to be the cells located nearest in the geometrical sense. Cellular automata were originally introduced by von Neumann and Ulam [7] as idealization of biological self-production.

Table 5 A Two-Input Transition Function

Current state	Current left input	Current right input	Next state
0	0	0	?
0	0	1	?
0	1	0	?
0	1	1	?
1	0	0	?
1	0	1	?
1	1	0	?
1	1	1	?

Cellular automata show the interesting property that they yield quite diverse and interesting behavior. Actually, Wolfram [18] systematically investigated all possible transition functions of one-dimensional cellular automata. He found out that there exist four types of cellular automata that differ significantly in their behavior; (1) automata where any dynamic soon dies out, (2) automata that soon come to periodic behavior, (3) automata that show chaotic behavior, and (4), the most interesting ones, automata whose behaviors are unpredictable and nonperiodic but that show interesting, regular patterns.

Conway's Game of Life in its original representation in *Scientific American* [10] can serve as a fascinating introduction to the ideas involved. The game is framed within a two-dimensional cell space structure, possibly of infinite size. Each cell is coupled to its nearest physical neighbors both laterally and diagonally. This means for a cell located at point (0, 0) its lateral neighbors are at (0, 1), (1, 0), (0, −1), and (−1, 0) and its diagonal neighbors are at (1, 1), (1,−1), (−1, 1) and (−1, −1), as shown in Fig. 3. The neighbors of an arbitrary cell at (i, j) are the cells at $(i, j+1)$, $(i+1, j)$, $(i, j-1)$, $(i-1, j)$, $(i+1, j+1)$, $(i+1, j-1)$, $(i-1, j+1)$, and $(i-1, j-1)$, which can be computed from the neighbors at (0,0) by a simple translation. The *state* set of a cell consists of one variable, which can take on only two values, 0 (*dead*) and 1 (*alive*). Individual cells survive (are alive and stay alive), are born (go from 0 to 1), or die (go from 1 to 0) as the game progresses. The rules as defined by Conway are as follows:

1. A live cell remains alive if it has between 2 and 3 live cells in its neighborhood.
2. A live cell will die because of overcrowding if it has more than 3 live cells in its neighborhood.
3. A live cell will die because of isolation if it has fewer than 2 live neighbors.
4. A dead cell will become alive if it has exactly 3 alive neighbors.

When started from certain configurations of alive cells, the Game of Life shows interesting behavior over time. As a state trajectory evolves, live cells form varied and dynamically changing clusters. The idea of the game is to find new patterns and study their behavior. Figure 4 shows some interesting patterns.

The Game of Life exemplifies some of the concepts introduced earlier. It evolves on a *discrete time base* (time advances in steps 0,1,2,. . .) and is a *multicomponent system* [it is composed of *components* (cells) that are coupled

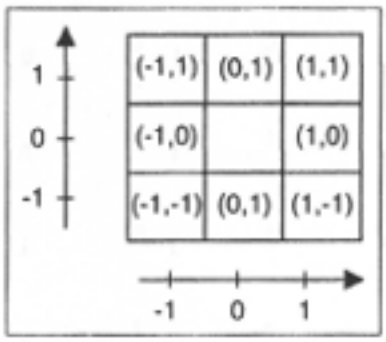

Figure 3 Cellular coupling structure for Game of Life.

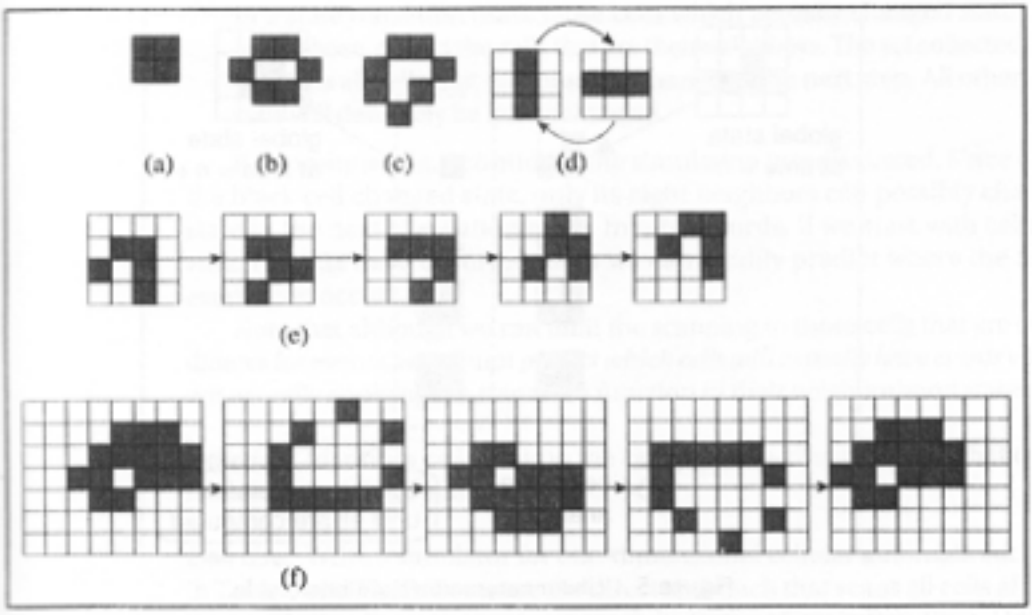

Figure 4 Patterns from Conway's Game of Life: Patterns (a) to (c) are stable—they don't change; (d) is an oscillating patterns; (e) and (f) are cycles of patterns that move.

together]. In contrast to the *local* state (the state of a cell), the *global state* refers to the collection of states of all cells at any time. In the Game of Life, this is a finite pattern, or configuration, of alive cells with all the rest being dead. Every such state starts a *state trajectory* (sequence of global states indexed by time) that either ends up in a cycle or continues to evolve forever.

Exercise: If an initial state is a finite configuration of alive cells, why are all subsequent states also finite configurations?◆

3.2.3 Cellular Automaton Simulation Algorithms

The basic procedure for simulating a cellular automaton follows the discrete time simulation algorithm introduced earlier. That is, at every time step we scan all cells, applying the state transition function to each, and saving the next state in a second copy of the global state data structure. When all next states have been computed, they constitute the next global state and the clock advances to the next step. For example, let's start with the three live cells in a triangle shown in the upper left of Fig. 5. Analyzing the neighborhood of the lower corner cell, we see it is alive and has 2 alive neighbors; thus, it survives to the next generation. As shown, the other alive cells also survive. However, only one new cell is born. This is the upper right corner cell, which has exactly three neighbors.

Of course, as stated, the cell space is an infinite and we can't possibly scan all the cells in a finite amount of time. To overcome this problem, the examined part of the space is limited to a finite region. In many case, this region is fixed throughout the simulation. For example, a two-dimensional space might be represented by a square array of size N (= 1,2,. . .). In this case, the basic algorithm scans all N^2 cells at every time step. Something

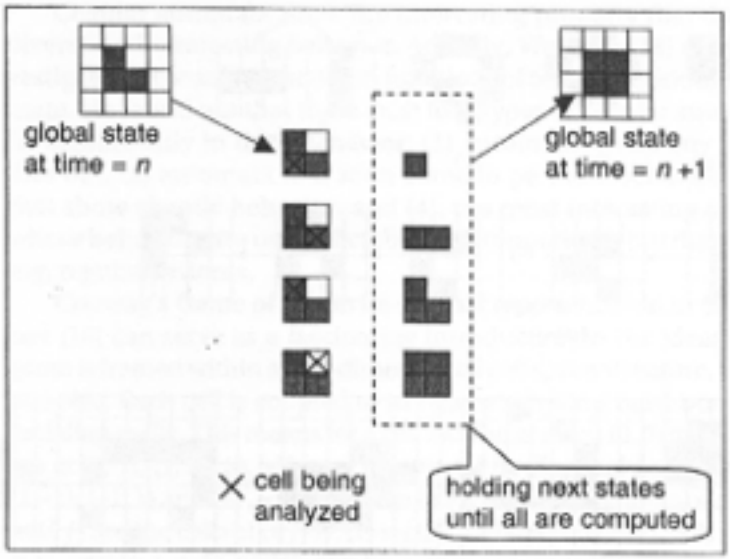

Figure 5 Cellular automaton simulation cycle.

must be done to take care of the fact that cells at the boundary lack their full complement of neighbors. One solution is to assume fixed values for all boundary cells (for example, all dead). Another is to wrap the space in toroidal fashion (done by letting the index N also be interpreted as 0). But there is a smarter approach that can handle a potentially infinite space by limiting the scanning to only those cells that can potentially change states at any time. This is the discrete event approach discussed next.

3.2.4 Discrete Event Approach to Cellular Automaton Simulation

In discrete time systems, at every time step each component undergoes a "state transition"; this occurs whether or not its state actually changes. Often, only a small number of components really change. For example, in the Game of Life, the dead state is called a *quiescent* state—if the cell and all its neighbors are in the quiescent state, its next state is also quiescent. Since most cells in an infinite space are in the quiescent state, relatively few actually change state. Put another way, if *events* are defined as changes in state (e.g., births and deaths in the Game of Life), then often there are relatively few events in the system. Scanning all cells in an infinite space is impossible, but even in a finite space, scanning all cells for events at every time step is clearly inefficient. A discrete event simulation algorithm **concentrates on processing events** rather than cells and is inherently more efficient.

The basic idea is to try to predict whether a cell will possibly change state or will definitely be left unchanged in a next global state transition. Suppose we have a set of potentially changing cells. Then we only examine those cells in the set. Some or all of these cells are observed to change their states. Then, in the new circumstances, we collect the cells that can possibly change state in the next step. The criterion under which cells can change is simple to define. *A cell will not change state at the next state transition time, if none of its neighboring cells changed state at the current state transition time.* The event-based simulation procedure follows from this logic:

In a state transition mark those cells which actually changed state. From those, collect the cells that are their neighbors. The set collected contains all cells that can possibly change at the next step. All other cells will definitely be left unchanged.

For example, Fig. 6 continues the simulation just discussed. Since only the black cell changed state, only its eight neighbors can possibly change state in the next simulation cycle. In other words, if we start with cells at which events have occurred, then we can readily predict where the next events can occur.

Note that although we can limit the scanning to those cells that are candidates for events, we cannot *predict which cells will actually have events* without actually applying the transition function to their neighborhood states.

Exercise: Identify a cell that is in the can-possibly-change set but that does not actually change state.◆

Exercise: Write a simulator for one-dimensional cellular automata such as in Table 5 and Fig. 2. Compare the direct approach that scans all cells all the time with the discrete event approach in terms of execution time. Under what circumstances would it not pay to use the discrete event approach? (See later discussion in Chapter 16.)◆

Exercise: Until now we have assumed a neighborhood of eight cells adjacent to the center cell. However, in general, a neighborhood for the cell at the origin is defined as any finite subset of cells. This neighborhood is translated to every cell. Such a neighborhood is called reflection symmetric if whenever cell (i,j) belongs to it then so does (j,i). For neighborhoods that are not reflection symmetric, define the appropriate set of influences required for the discrete event simulation approach.◆

3.2.5 Switching Automata/Sequential Machines

Cellular automata are uniform both in their composition and interconnection patterns. If we drop these requirements but still consider connecting finite state components together, we get another useful class of discrete time models.

Switching automata (also called digital circuits) are constructed from flip-flop components and logical gates. The flip-flops are systems with

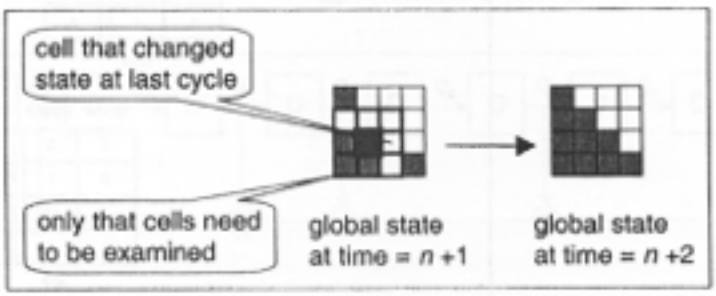

Figure 6 Collecting cells with possible events.

Table 6 State and Output Trajectories of a Delay Flip-Flop

Time	0	1	2	3	4	5	6	7	8	9	
Input trajectory	1	0	1	0	1	0	1	0	1	0	
State trajectory	0	1	0	1	0	1	0	1	0	1	0
Output trajectory	0	1	0	1	0	1	0	1	0	1	

binary states, the most straightforward example of which we have already met. This has the transition function shown in Table 1. However, instead of allowing the input to propagate straight through to the output, we will let the output be the current state (as in cellular automata). Table 6 shows how the output trajectory now lags the input trajectory by one time step. The difference, we'll see later, is between so-called Mealy and Moore models of sequential machines. In Mealy networks the effects of an input can propagate throughout space in zero time—even cycling back on themselves, causing "vicious circles" or ill-behaved systems.

To construct networks we can not only couple flip-flop outputs to inputs, but also connect them through so-called *gates*, which realize elementary Boolean functions. As we shall see later, these are instantaneous or memoryless systems. As shown in Fig. 7, their outputs are directly deter-

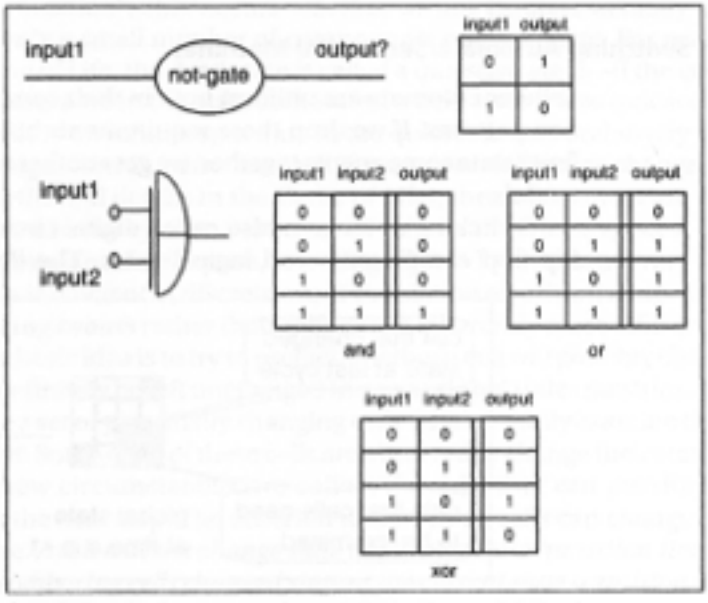

Figure 7 Gates realizing elementary boolean functions.

mined by their inputs with no help from an internal state. The clock usually used in synchronous machines determines a constant time advance. This enables transistor circuits to be adequately represented by discrete time models, at least for their functional behavior.

The switching automaton in Fig. 8 is made up of several flip-flop components coupled in series and a feedback function defined by a coupling of XOR gates. Each flip-flop is an elementary memory component as before. Flip-flops q_1 to q_7 are coupled in series, that is, the state of flip-flop i defines the input and hence the next state of flip-flop i–1. This linear sequence of flip-flops is called a *shift register*, since it shifts the input at the first component to the right each time step. The latter input may be computed using gates. For example, the input to flip-flop 8 is defined by an XOR-connection of memory values q_i and input value x: $x \oplus q_8 \oplus q_6 \oplus q_5 \oplus q_3 \oplus q_1$. (An XOR, $q_1 \oplus q_2$ outputs the exclusive-or operation on its inputs, i.e., the output is 0 if, and only if, both inputs are the same, otherwise 1.)

In Chapter 8 we will discuss simulators for such combinations of memoryless and memory-based systems.

3.2.6 Linear Discrete Time Networks and Their State Behavior

Fig. 9 illustrates a Moore network with two delay elements and two memoryless elements. However, in distinction to the switching automaton above, this network now is defined over the reals. The structure of the network can be represented by the matrix

$$\begin{bmatrix} 0 & g \\ -g & 0 \end{bmatrix},$$

where g and $-g$ are the gain factors of the memoryless elements. Started in a state represented by the vector [1,1], (each delay is in state 1), the next state $[q_1, q_2]$ of the network is $[-g, g]$, which can be computed by the matrix multiplication

$$\begin{bmatrix} g \\ -g \end{bmatrix} = \begin{bmatrix} 0 & g \\ -g & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

The reason that a matrix representation and multiplication can be used is that the network has a *linear* structure. This means that all components produce outputs or states that are linear combinations of their inputs and states. A delay is a very simple linear component—its next state is identical to its input. A memoryless element with a gain factor is called

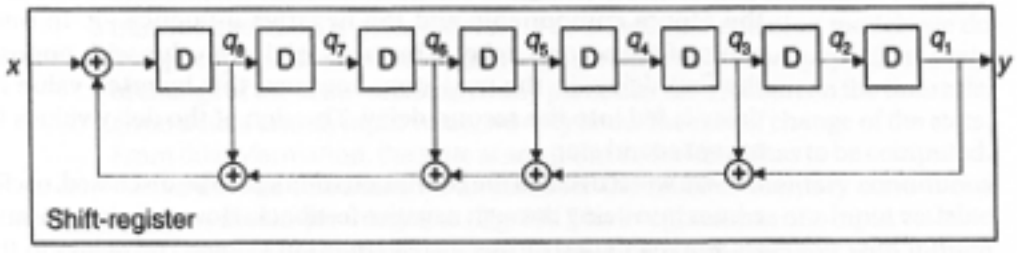

Figure 8 Shift register coupling flip-flop elements and logical gates.

a *coefficient* element and it multiplies its current input by the gain to produce its output. Both of these are simple examples of linearity. A summer, that is a memoryless function that adds its inputs to obtain its output, is also a linear element.

A Moore discrete time system is in *linear matrix* form if its transition and output functions can be represented by matrices (A,B,C). This means that its transition function can be expressed as

$$\delta(q,x) = Aq + Bx.$$

Here q is a real n-dimensional state vector, and A is an n by n matrix. Similarly, x is a real m-dimensional input vector, and B has dimension m by n. Also, the output function is

$$\lambda(q) = Cq,$$

where, if the output y is a p-dimensional vector, then C is an n by p-dimensional matrix. A similar definition holds for Mealy discrete time system.

Exercise: Any discrete time network with linear structure can be represented by a linear system in matrix form. Conversely, any linear discrete time system in matrix form can be realized by a discrete time network with linear structure. Develop procedures to convert one into the other.◆

The state trajectory behavior of the linear system in Fig. 9 is obtained by interatively multiplying the matrix A into successive states. Thus, if $[g,-g]$ is the state following $[1,1]$, then the next state is

$$\begin{bmatrix} -g^2 \\ g^2 \end{bmatrix} = \begin{bmatrix} 0 & g \\ -g & 0 \end{bmatrix} \begin{bmatrix} g \\ -g \end{bmatrix}$$

Thus, the state trajectory starts out as

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \ldots \begin{bmatrix} g \\ -g \end{bmatrix} \ldots \begin{bmatrix} -g^2 \\ g^2 \end{bmatrix}.$$

Exercise: Characterize the remainder of the state trajectory.◆

There are three distinguishable types of trajectories as illustrated in Fig. 9. When $g = 1$, we have a steady alternation between +1 and −1. When $g < 1$, the envelope of values decays exponentially; and finally, when $g > 1$, the envelope increases exponentially.

Actually the model just given is a discrete form of an oscillating system. In the following sections we will learn to understand its better-known continuous system counterpart. In the discrete form, oscillations (switching from positive to negative values) come into being because of the delays in the Moore components and the negative influence $-g$. In one step the value of the second delay is inverted, giving a value with opposite sign to the first delay. In the next step, however, this inverted value in the first delay is fed into the second delay. The sign of the delay values is inverted every second step.

We will see that in the continuous domain to be discussed, oscillation also comes into being through negative feedback. However, the continuous models work with derivatives instead of input values. The delays in the discrete domain correspond to integrators in the continuous domain. But let us first

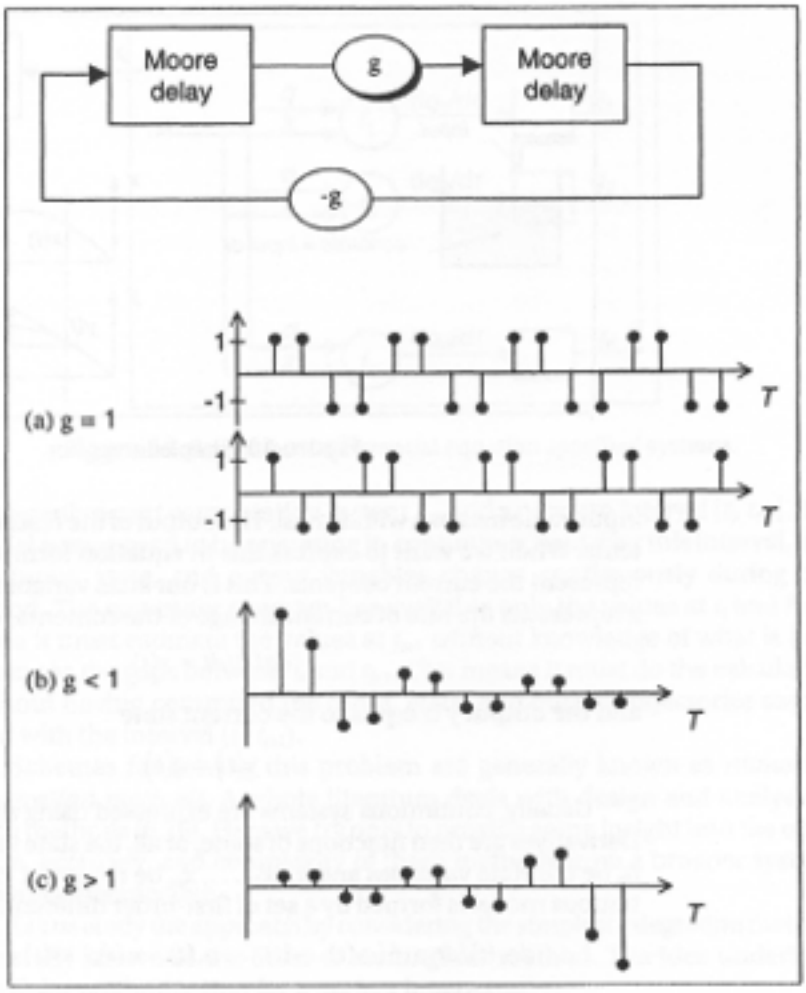

Figure 9 Simple linear Moore network and its behavior.

discuss the basics of modeling and simulation of continuous systems, and then come back to compare the two forms of oscillating systems thereafter.

3.3 Differential Equation Models and Their Simulators

In discrete time modeling we had a state transition function that gave us the information of the state at the next time instant given the current state and input. In the classical modeling approach of differential equations, the state transition relation is quite different. For differential equation models we do not specify a next state directly, but use a *derivative function* to specify the rate of change of the state variables. At any particular time instant on the time axis, given a state and an input value, we only know the rate of change of the state. From this information, the state at any point in the future has to be computed.

To discuss this issue, let us consider the most elementary continuous system—the simple integrator (Fig. 10). The integrator has one input variable x and one output variable y. One can imagine it as a reservoir with infinite capacity. Whatever is put into the reservoir is accumulated—but a negative

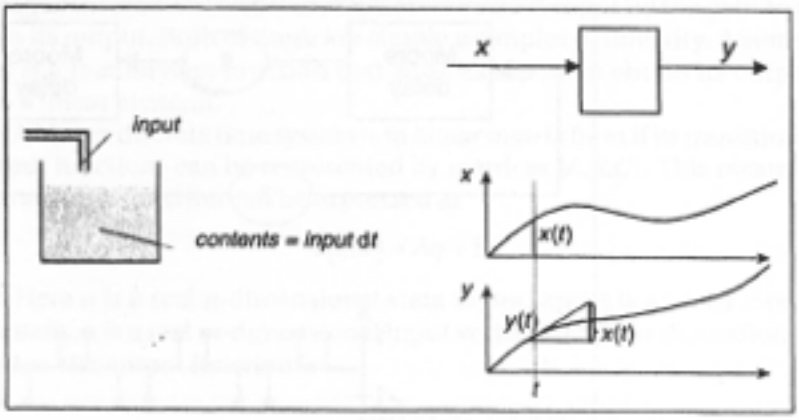

Figure 10 Simple integrator.

input value means a withdrawal. The output of the reservoir is its current contents. When we want to express this in equation form we need a variable to represent the current contents. This is our state variable q. The current input x represents the rate of current change of the contents, which we express by

$$dq(t)/dt = x(t),$$

and the output y is equal to the current state

$$y(t) = q(t).$$

Usually, continuous systems are expressed using several state variables. Derivatives are then functions of some, or all, the state variables. Let $q_1, q_2, \ldots, q_n$ be the state variables and $x_1, x_2, \ldots, x_m$ be the input variables. Then a continuous model is formed by a set of first-order differential equations,

$$dq_1(t)/dt = f_1(q_1(t), q_2(t), \ldots, q_n(t), x_1(t), x_2(t), \ldots, x_m(t))$$
$$dq_2(t)/dt = f_2(q_1(t), q_2(t), \ldots, q_n(t), x_1(t), x_2(t), \ldots, x_m(t))$$
$$\ldots$$
$$dq_n(t)/dt = f_n(q_1(t), q_2(t), \ldots, q_n(t), x_1(t), x_2(t), \ldots, x_m(t)).$$

Note that the derivatives of the state variables q_i are computed, respectively, by functions f_i that have the state and input vectors as arguments. This can be shown in diagrammatic form as in Fig. 11. The state and input vector are input to the rate of change functions f_i. Those provide as output the derivatives dq_i/dt of the state variables q_i, which are forwarded to integrator blocks. The outputs of the integrator blocks are the state variables q_i.

3.3.1 Continuous System Simulation

Figure 11 reveals the fundamental problem that occurs when a continuous system is simulated on a digital computer. In the diagram we see that given a state vector q and an input vector x for a particular time instant t_i we only obtain the derivatives dq_i/dt. But how do we obtain the dynamic behavior of the system over time? In other words, how do we obtain the state values after this time? This problem is depicted in Fig. 12. In digital simulation, there is

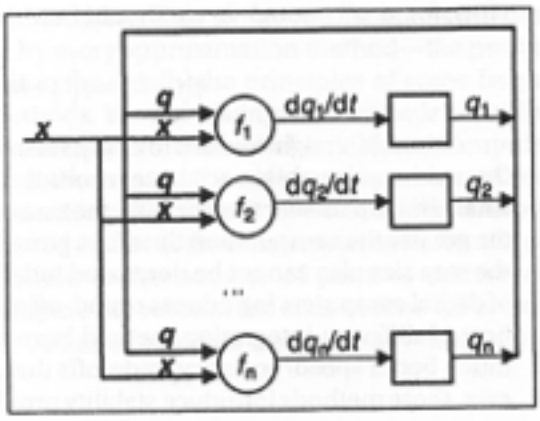

Figure 11 Structure of differential equation specified systems.

necessarily a next computation instant t_{i+1} and a nonzero interval $[t_i, t_{i+1}]$. The model is supposed to be operating in continuous time over this interval, and the input, state, and output variables change continuously during this period. The computer program has available only the values at t_i and from those it must estimate the values at t_{i+1} *without* knowledge of what is happening in the gaps between t_i and t_{i+1}. This means it must do the calculation without having computed the input, state, and output trajectories associated with the interval (t_i, t_{i+1}).

Schemes for solving this problem are generally known as *numerical integration methods*. A whole literature deals with design and analysis of such methods [6,16]. Here we try only to provide some insight into the operation, accuracy, and complexity of these methods from a broader systems theoretic perspective.

Let us study the approach by considering the simplest integration method, generally known as the *Euler* or *rectangular* method. The idea underlying the Euler method is that for a perfect integrator

$$\frac{dq(t)}{dt} = \lim_{h \to 0} \frac{q(t+h) - q(t)}{h}.$$

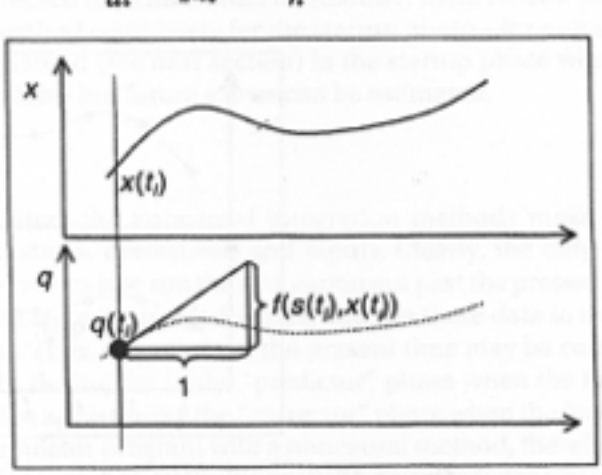

Figure 12 Continuous system simulation problem.

Thus, for small enough h, we should be able to use the approximation

$$q(t+h) = q(t) + h \cdot \frac{dq(t)}{dt}.$$

Although straightforward to apply, Euler integration has some drawbacks. On one hand, to obtain accurate results the step size h has to be sufficiently small. But the smaller the step size, the more iterations are needed, and hence the greater the computation time for a given length of run. On the other hand, the step size also cannot be decreased indefinitely, since the finite word size of digital computers introduces round-off and truncation errors. Therefore, a host of different integration method have been developed that often show much better speed/accuracy trade-offs than the simple Euler method. However, these methods introduce stability problems, as we discuss in a moment.

Exercise: Show that the number of iterations varies inversely to the step size in Euler integration. How does reducing the step size an order of magnitude affect the number of iterations?◆

The basic idea of numerical integration is easily stated—an integration method employs estimated past and/or future values of states, inputs, and derivatives in an effort to better estimate a value for the present time (Fig. 13). Thus, computing a state value $q(t_i)$ for a present time instance t_i may involve computed values of states, inputs, and derivatives at prior computation times $t_{i-1}, t_{i-2}, \ldots$ and/or predicted values for the current time t_i and subsequent time instants $t_{i+1}, t_{i+2}, \ldots$,

$$q(t_i) = integration\_method\big(\ldots, q(t_{i-2}), f\big(q(t_{i-2}), x(t_{i-2})\big),$$
$$q(t_{i-1}), f\big(q(t_{i-1}), x(t_{i-1})\big), q'(t_i), f\big(q'(t_i), x'(t_i)\big), q'(t_{i+1}),$$
$$f\big(q'(t_{i+1}), x'(t_{i+1})\big), \ldots\big)$$

where q' and x' are the state and input estimates.

Notice that the values of the states and the derivatives are mutually interdependent—the integrator itself causes a dependence of the states on the derivatives and through the derivative functions f_i the derivatives are

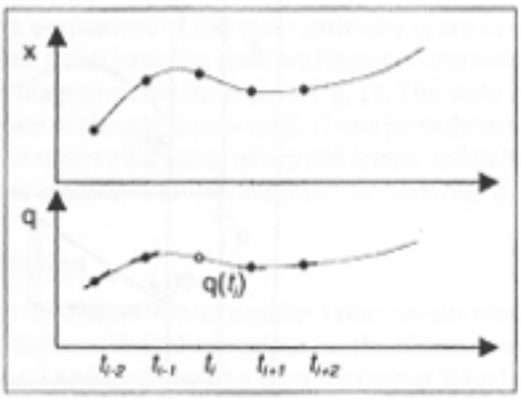

Figure 13 Computing state values at time t_i based on estimated values at time instants prior and past time t_i.

dependent on the states. This situation sets up an inherent difficulty that must be faced by every approximation method—the propagation of errors.

We now describe briefly the principles of some frequently employed integration methods. We can distinguish methods according to whether or not they use estimated future values of variables to compute the present values. We call a method *causal* if it only employs values at prior computation instants. A method is called *non-causal* if in addition to prior values it also employs estimated values at time instants at and after the present time. The order of a method is the number of pairs of derivative/state values it employs to compute the value of the state at time t_i. For example, a method employing the values of q and its derivative at times t_{i-2}, t_{i-1}, t_i, and t_{i+1} is of order 4.

Causal Methods

A causal method of order m is often a linear combination of the state and derivative values at time instants t_{i-m} to t_{i-1} with coefficients chosen to minimize the error from the computed estimate to the real value. Usually the basis of the derivation of the coefficients is the assumption that the state trajectories and the output trajectories are smooth enough to be approximated by the dth-order polynomial. Of course, often it is not known beforehand whether the requisite smoothness indeed obtains.

Example: The Adams method is a simple causal method of order 2 and can be described by

$$q(t_i) = q(t_{i-1}) + h\left(3f\left(q(t_{i-1}), x(t_{i-1})\right) - f\left(q(t_{i-2}), x(t_{i-2})\right)\right).$$

A general problem in causal integration methods is the startup, namely, the determination of the state, derivative, and output values for the times t_{1-m} to t_0 at the initial simulation time t_1. This is generally referred to as the *startup problem*. A solution to this problem is to use causal methods of lower orders in the startup phase. That is, in the first cycle, where only the initial state is available, a method of order 1 is used and subsequently the order of the method is increased to its final order m. Another, more reliable solution is to change the method completely for the startup phase—for example, to use a noncausal method (see next section) in the startup phase when past values are not available but future values can be estimated.

Noncausal Methods

As we have indicated, the noncausal integration methods make use of "future" values of states, derivatives, and inputs. Clearly, the only way of obtaining "future" values is to run the simulation out past the present model time, calculate and store the needed values, and use these data to estimate the present values. Thus, the values at the present time may be computed twice—tentatively, during the initial "predictor" phase when the future is simulated, and once again during the "corrector" phase when the final value is computed. A simulator program with a noncausal method, therefore, has to work in several phases. At first there may be several predictor phases, and finally there is usually one corrector phase.

Example: A simple predictor–corrector method generally known as the *Heun* method is

$$q'(t_i) = q(t_{i-1}) + hf\left(q(t_{i-1}), x(t_{i-1})\right)$$
$$q(t_i) = q(t_{i-1}) + h/2\left(f\left(q'(t_i), x'(t_i)\right) + f\left(q(t_{i-1}), x(t_{i-1})\right)\right),$$

where $q'(t_i)$ is the predicted value and $q(t_i)$ is the final corrected value.

In some methods, called *variable stepsize* methods, the difference between two computed values is compared with a criterion level; if the estimated error is too large, the step size is decreased and the predictor–corrector cycle repeated until a step size is found for which the difference is below the criterion level. Of course, if the state trajectory is not sufficiently smooth, no finite step size may be found for which the divergence is sufficiently small, and the simulation will drive to an expensive halt.

Many noncausal methods, among them the most popular *Runge–Kutta* methods, do not use past values but only use the current state and input value to make the predictions of future values. For those methods, the startup phase is not a problem. Such methods therefore are also good alternatives to employ in the startup phase of causal methods.

To Euler or Not: That Is The Question

Error arises from two sources: Some error is introduced in each step by the approximation method, and some accumulates through the effect of prior error propagating through the system. In case of integration methods, the first source of error exists even if the values of state and derivative are correct. The second source of error is due to the mutual dependence of states and derivatives just mentioned. An error in state is transmitted to the derivative functions, where it may further affect the state values. An integration method may tend to amplify or dampen the effect of error feedback. All other things being equal, a so-called *stable* method (one that dampens error propagation) is preferable to one that amplifies the error propagation. But the error propagation does not only depend on the method, it also depends on the nature of the model—whether it tends to amplify or dampen deviations (e.g., whether trajectories emerging from initial states close together tend to remain close together). From this we can see that the choice of integration method, or for that matter any simulation algorithm, for a given model is not an easy one.

We have suggested that multipoint methods can be faster than direct Euler integration while preserving accuracy. However, these methods also have their drawbacks. They introduce stability and startup problems that Euler doesn't suffer from. They make assumptions about the analytic nature of the trajectories that can easily be violated in hybrid models with state events (see Chapter 9). Also, with modern high-speed computers, the use of Euler and other direct integration methods is becoming more practical.

Fortunately, modern environments and packages for continuous simulation shield the modeler from many of these considerations. Indeed, in principle, users should be able to work with models independently of the underlying integration method. Unfortunately, however, today's environments are not perfect and the user has to be on the lookout for signs that the simulator

is not faithfully generating the model trajectories. For example, puzzling results may be due to the integration becoming unstable rather than an interesting model behavior. Or more perniciously, innocent-looking simulated trajectories may hide unsuspected integration errors. Table 7 summarizes the pros and cons of integration methods.

The problems of error introduction and error propagation are endemic to simulator and model construction throughout the whole M&S enterprise. We will take up these issues later in Chapters 13 and 14.

3.3.2 Feedback in Continuous Systems

After acquiring a basic understanding of differential equation models and how they are simulated on digital computers, let us now try to get some insight into the nature of continuous system behavior. By considering several elementary systems we will demonstrate that the reason for the emergence of quite complex behavior is *feedback*. In a system model embodying feedback loops, state variables are fed back to influence their own rates of change. Feeding back a state variable to its own derivative can be direct or also involve several other state variables. Figure 14 shows a feedback loop where a state variable q_1, influences the derivative of q_2, which again influences q_3, and so on. Finally, a state variable q_n is used to define the derivative of q_1, closing the feedback loop.

Qualitative analysis of the feedback loops in a system can give insights into its possible behaviors. Most important is whether a feedback loop is positive or negative. When traversing the feedback loop, we can count the signs of the direct influences between the state variables. When the sign of a function f_i is positive, we speak of a positive influence from q_{i-1} to q_i, which means that a positive value of q_{i-1} will make q_i increase. Conversely, when the sign of a function f_i is negative, a positive value of q_{i-1} will cause a decrease of q_i. A positive feedback loop is one with a even number of negative influences. A negative feedback loop is one with an odd number of negative influences.

How do they differ? Let us consider as an example a positive feedback loop with zero negative influences and a negative feedback loop with one negative influence. In the positive feedback loop a greater value of a state variable will cause all influenced variables in the feedback loop to increase.

Table 7 Summary of Integration Method Speed–Accuracy Trade-Offs

Integration method	Efficiency	Startup problems	Stability problems	Robustness
Euler	Low	None	Not for sufficiently small step sizes	High
Causal methods	High	Yes	Yes	Low
Noncausal methods	High	No, if only future values used	Yes	Low

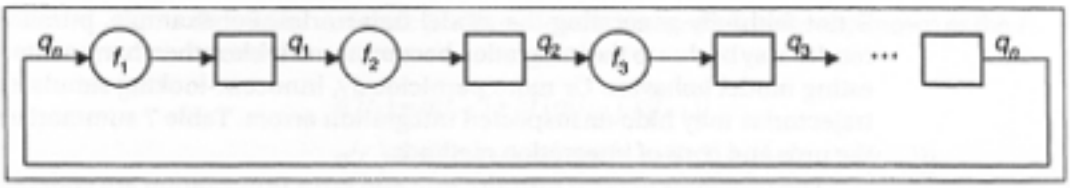

Figure 14 Feedback loop.

In turn, this will cause its own derivative to increase and the variable will get even greater. This feedback loop causes the variable to grow indefinitely. In some contexts, this is called unstable behavior and is undesirable. However, in other contexts, such as biological and economic growth, this is very desirable (at least for some part of the trajectory). For negative feedback loops, however, the negative influence will tend to stabilize a system. Growth of the state variable will tend to be turned into negative influence on its derivative, which would cause the state variable to decrease (we say "tend" because, the situation is actually more complex; see Ref. [13]).

Exercise: Argue why a feedback loop with an even number of negative influences works like one with no negative influence and one with an odd number of negative influences is the same as one with one negative influence.◆

3.3.3 Elementary Linear Systems

After this general discussion of feedback in continuous systems, let us introduce some well-known basic continuous systems and study their behavior. These often appear as elementary substructures in models of complex real phenomena independently of the application domain at hand (the feedback takes different forms in the different structures). As we will see, simple continuous systems can show quite interesting behavior, despite their simple-looking structure. The reason for this is feedback.

First of all, let us discuss a system with one state variable with a direct feedback loop to itself. The derivative of the state variable is computed by the state variable multiplied with a linear factor c. When the factor c is positive, we have a positive feedback. In this case, the system will grow exponentially and we speak of an *exponential growth*. Conversely, when the factor c is negative, we have a negative feedback and the variable will shrink approaching zero exponentially. We have a *exponential decay*. When the factor is zero, the state remains constant. Figure 15 shows the system structure and several different behaviors for different values for factor c. The larger c is in magnitude, the steeper the curve, whether increasing or decreasing.

As illustrated in Fig. 16, when we apply an input to an exponential decay, we have the so-called *exponential delay of first order*. This type of continuous system plays an important role in many real-world phenomena. In abstract terms, this is the right type of model to use for system with a driving force and a linear damping. The input to the system represents the driving force and the feedback through negative factor c models the damping. These two are added, defining the derivative for the state variable. The characteristic of the system is that the system reaches an equilibrium state when the damping through the

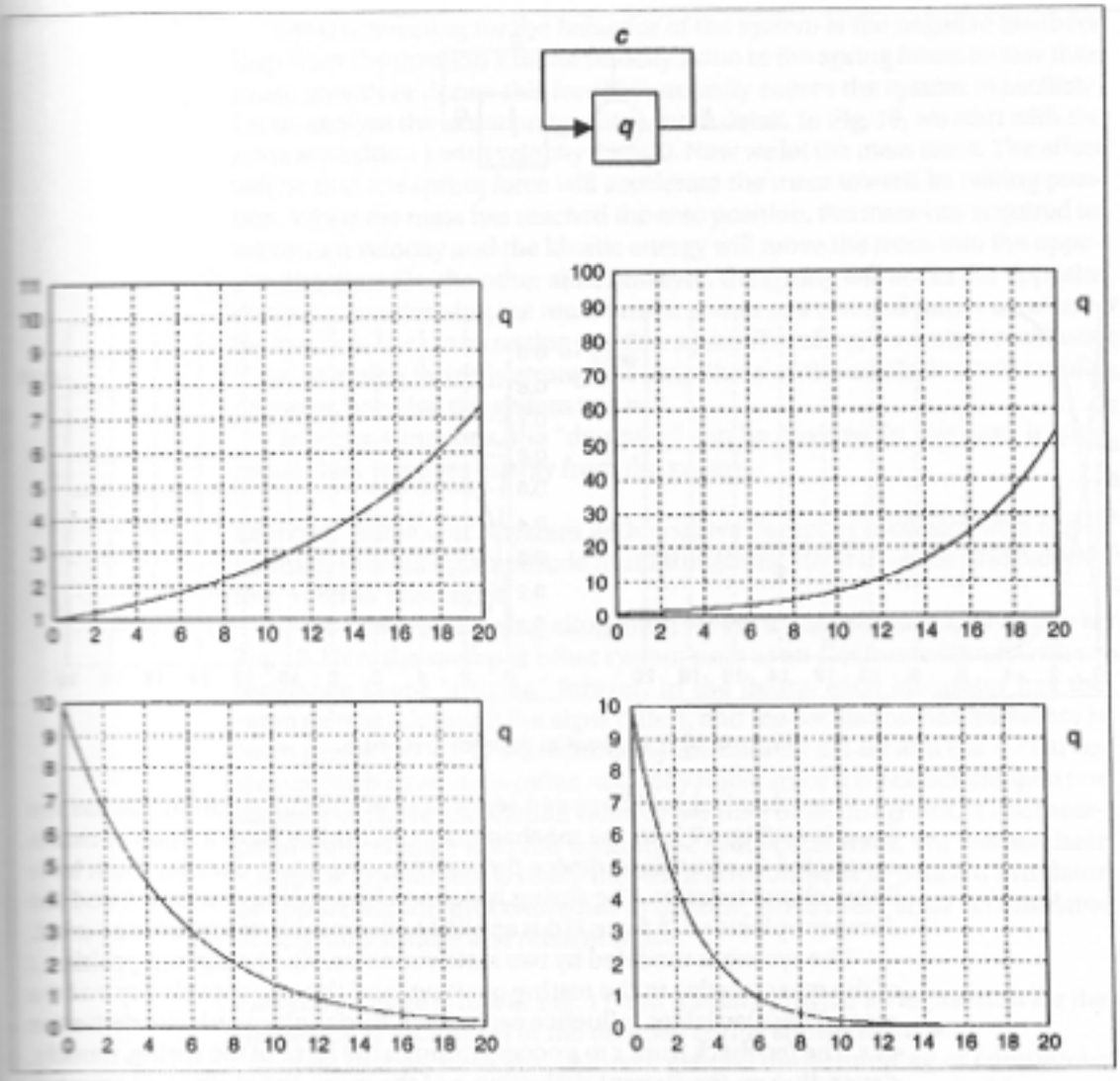

Figure 15 Exponential growth and decay.

feedback $-c * q$ equals the driving input. Figure 16 shows this for a constant input force x_1 and different damping factors c. The greater the damping factor, the smaller the equilibrium state.

There are many real-world phenomena that can be modeled by an exponential delay of first order. Examples range from heating systems where the input is the heat supply and the damping represents the heat losses to the environment, to mechanical systems where the input is the force supplied and the damping represents the friction that should be counteracted, to input of pollution into a natural system and its reduction through an absorption process.

The examples just given are systems with one state variable with a direct feedback loop to itself. Let us now consider a system in two variables. The *linear (or harmonic) oscillator of second order* is the best known example where the effect of feedback can easily be studied.

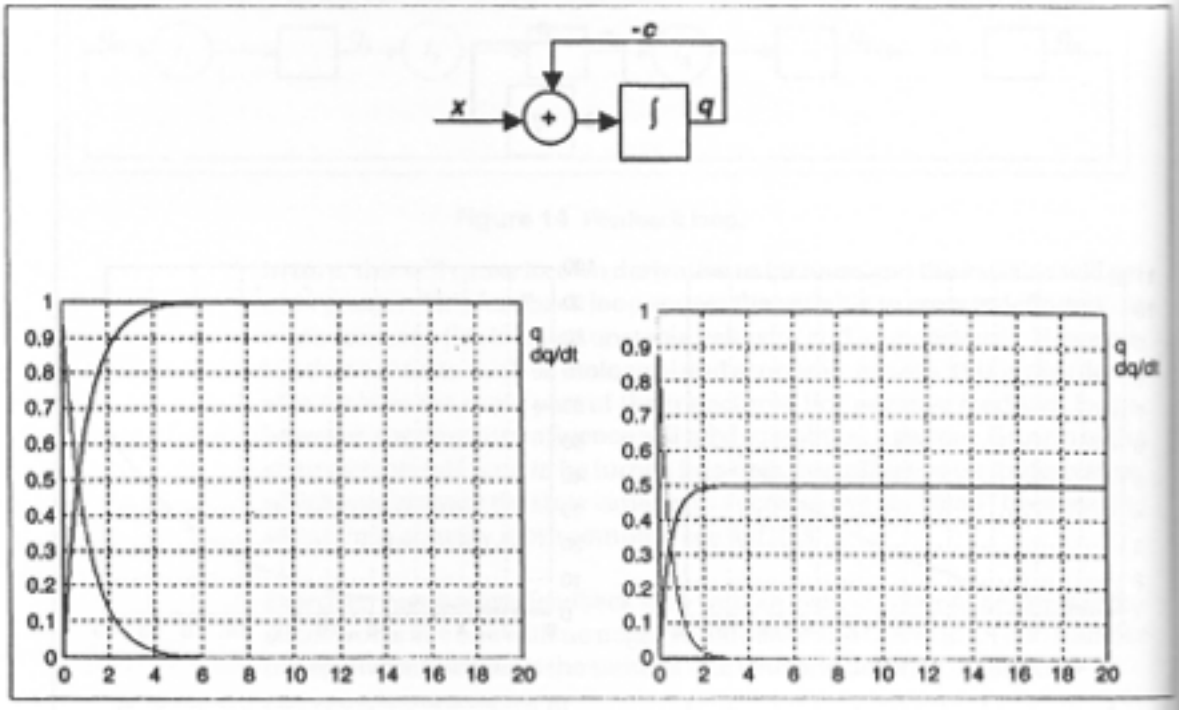

Figure 16 Exponential delay of first order.

A mechanical oscillator can serve as a reference system to discuss the equations (Fig. 17). In the mechanical oscillator we have a mass m that is connected to some fixed device through a spring and a damper. Both have linear characteristics. The spring is defined by a spring constant k and the damper by factor d. A force $F(t)$ is applied to the mass over some time period.

The system is modeled by two state variables, viz. the current position x of the mass relative to the resting position, and the current velocity v of the mass. These variables influence each other. By definition, v is the derivative of x. The feedback from x to v occurs through the force of the spring, namely, depending on the current deflection x of the mass, the spring will counteract and influence the velocity v by $-k * x$. Also, damping exerts a direct negative feedback from the velocity to itself ($d * v$). The greater the velocity, the greater the damping gets.

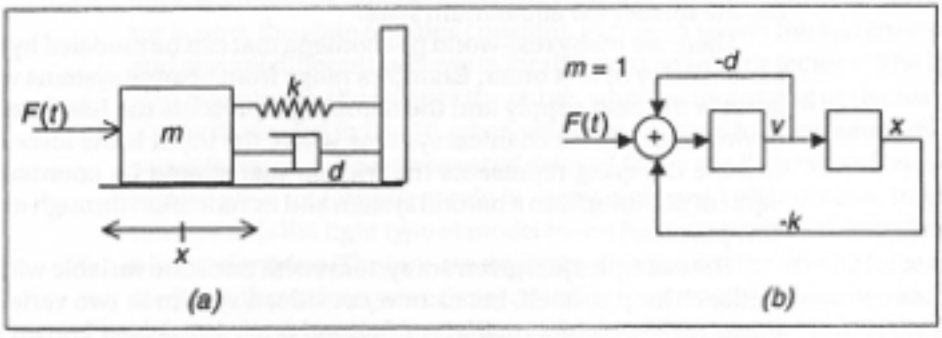

Figure 17 Damped linear oscillator of second order.

Most interesting for the behavior of the system is the negative feedback loop from the position x to the velocity v due to the spring force. Rather than cause growth or decay, this feedback actually causes the system to oscillate. Let us analyze the situation in a little more detail. In Fig. 18, we start with the mass at position 1 with velocity being 0. Now we let the mass loose. The effect will be that the spring force will accelerate the mass toward its resting position. When the mass has reached the zero position, the mass has acquired its maximum velocity and the kinetic energy will move the mass into the opposite direction. On the other side, however, the spring will act in the opposite direction, decelerating the mass until it comes to a rest and finally accelerating the mass back to its resting position again. Therefore, the system oscillates. If the damping factor is strong, the amplitude of the oscillation will rapidly decrease, bringing the system to a halt.

In some situations, the "damping" can be positive. In this case, it adds rather than removes energy from the system.

Exercise: Show that a system with positive damping oscillates with exponentially increasing amplitude (until something stops it—such as exploding into another system!).◆

If we remove damping altogether, we get a pure oscillator, as shown in Fig. 19. Here the spring or other system such as an electronic circuit with no resistance keeps "ringing" forever. In the figure, each integrator has the same gain, ω (although the signs differ), and the oscillation has frequency ω (with period $2\pi/\omega$). It is described by the familiar $\sin \omega t$ and $\cos \omega t$ curves shown. Such a system is called *neutrally* stable since it is neither damped nor undamped. If we add a small value to the state of an integrator, it will incorporate this change into the magnitude of its oscillation. We will see later (Chapter 16) that this is really the worst environment in which a simulator or approximation procedure has to operate, since every error accumulates in both an absolute and relative sense.

Exercise: Relate ω to k in Fig. 17 and thereby provide an expression for the frequency of oscillation in the damped spring shown there.◆

Now that we have also gained a general understanding of continuous modeling and simulation, let us come back to the comparison of the linear

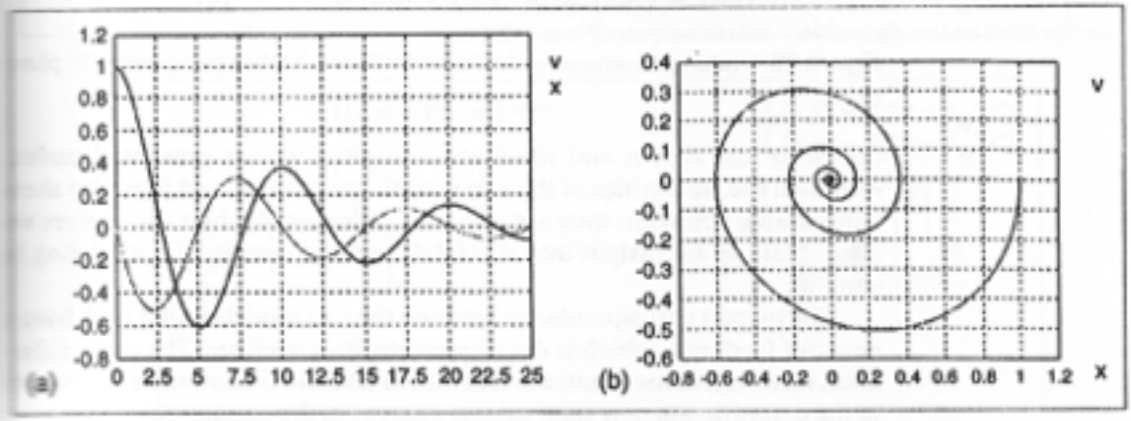

Figure **18** Damped linear oscillator: (a) position and velocity trajectories, (b) phase space.

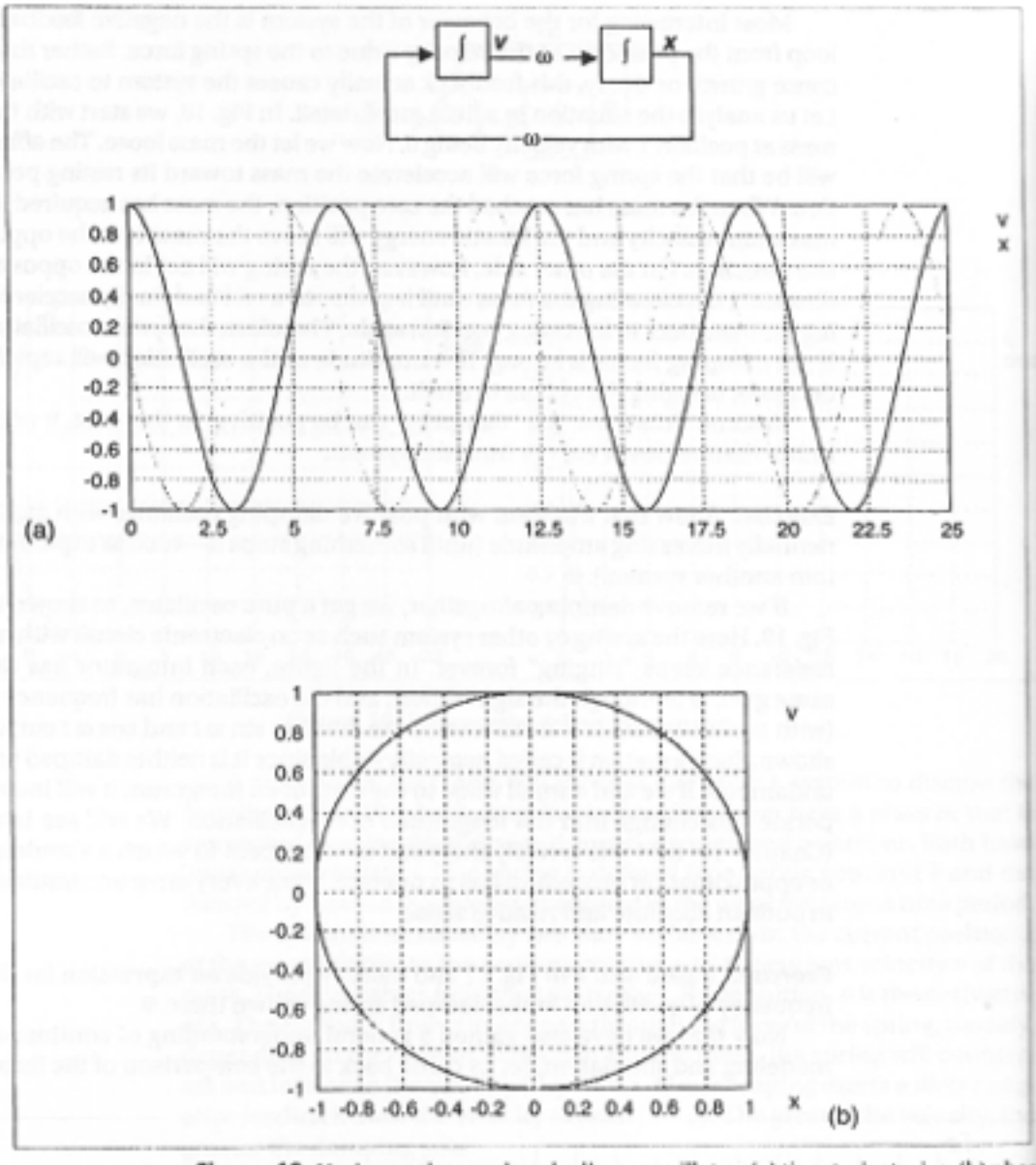

Figure 19 Undamped second-order linear oscillator: (a) time trajectories, (b) phase space.

oscillator just shown and the discrete time oscillator presented earlier. Although the similarities of the two models are obvious and they also show comparable behavior, they are also very different. By their differences we also obtain some insight into discrete time and continuous modeling in general.

The biggest correspondence between the two models is that both have a negative feedback, which is the reason why they oscillate. The main difference, however, comes from the fact that in discrete time models, the inputs to the delays define new state values, whereas in the continuous domain the inputs represent rate of change values for the state variables. Let us consider

the gains in the continuous and the discrete oscillators. In the discrete oscillator, the gain determines the amplification or decay of the value of one delay as it is fed to the other. In the continuous oscillator, the gain amplifies the rate of change of the influenced variable and thereby defines how *fast* the change of the state occurs. It determines the frequency of the oscillation. How is the frequency defined in the discrete domain? Actually by two things—namely, by the clock rate and by the number of delays in the feedback loop. Recall that in the discrete oscillator, we had two delays, and the sign of the values changed every two clock ticks. If we introduce a further delay, it will take another tick for a change in sign due to the negative feedback to propagate through all the delays. An oscillation frequency of three clock ticks results. Thus, the mechanism underlying the oscillation is quite different in the discrete and continuous cases.

3.3.4 Nonlinear Oscillators: Limit Cycles and Chaotic Behaviors

The systems we have considered so far were all linear. Now we discuss a nonlinear system—the so-called Van der Pol system, which comes into being when a nonlinear feedback is added to the linear oscillator. Instead of the linear damping factor $-d$, a nonlinear feedback is introduced that is dependent on the value of x—namely, the feedback is equal to $(1 - x^2) * v$. Figure 20a shows the model structure.

Through this nonlinear modification, the system acquires some quite astonishing properties. Independent of its initial state values, the system quickly comes into a stable oscillation. The reason for this is that for small values of x, the factor $(1 - x^2)$ is positive, thus leading to an amplification of the value v, whereas for large values of x, the factor $(1 - x^2)$ rapidly becomes negative, which leads to fast damping of v. The effect is that for small values of x, the feedback drives the values out of the region around zero, but as soon as the value is above 1 or −1, the factor becomes negative and drives the value of x back. The result is a stable oscillation, as shown in Fig. 21. Because the trajectory of the system always returns to a closed loop in state space, the latter is called a *limit cycle*.

Exercise: Compare cyclic behavior of the Van der Pol system with that of the second-order linear oscillator.◆

Lotka–Volterra predatory–prey systems form an interesting class of nonlinear models that exhibit oscillatory behavior—although not as limit cycles.

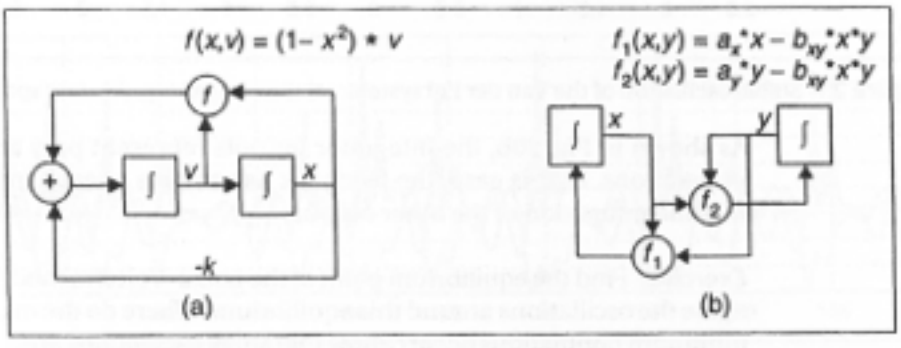

Figure 20 (a) Van der Pol and (b) Lotka–Volterra systems.

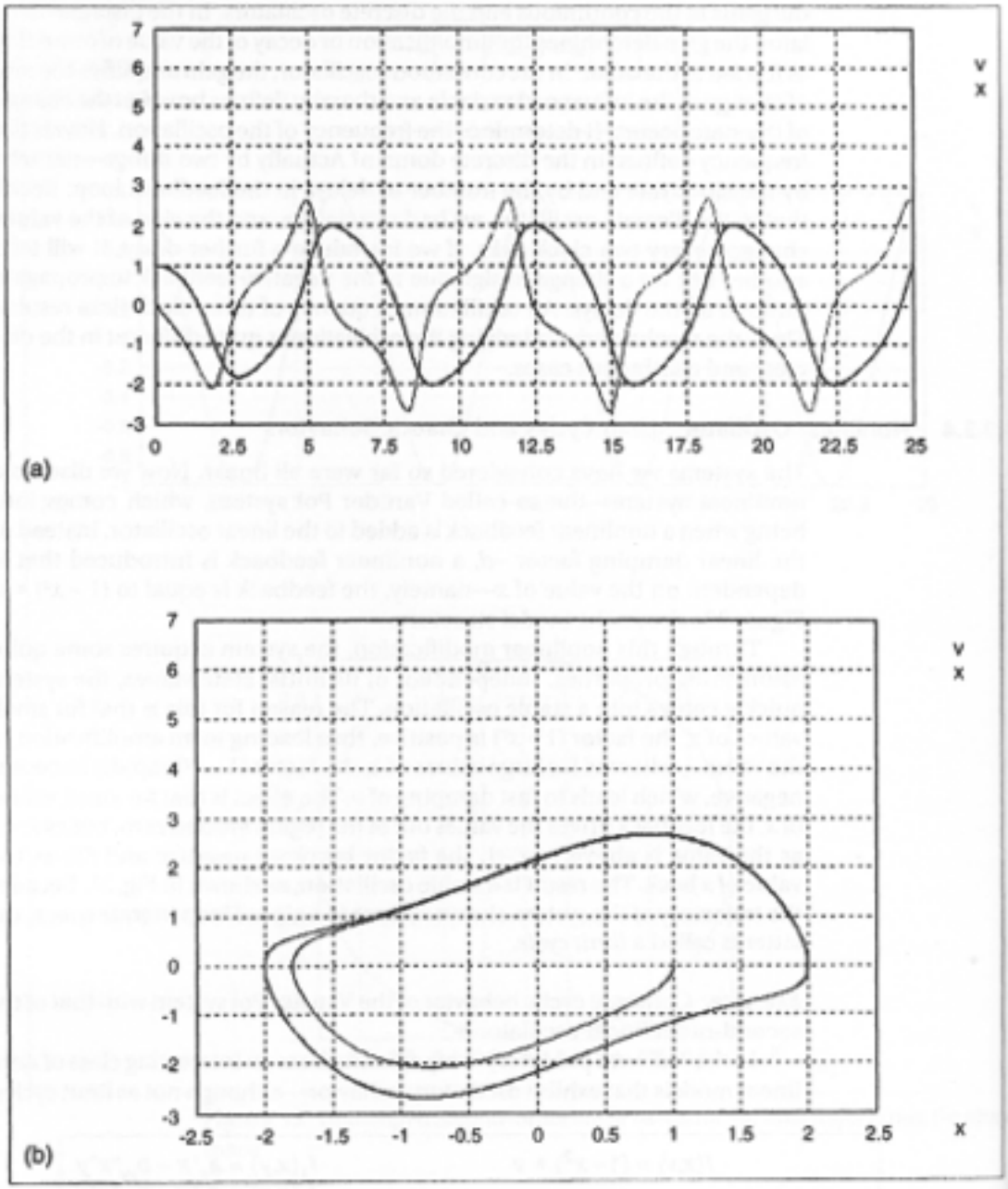

Figure 21 Stable oscillation of the Van der Pol system: (a) time trajectory, (b) state space portrait.

As shown in Fig. 20b, the integrator outputs represent prey and predator populations. In this case, the feedback parameters of each integrator is a nonlinear function of the other output.

Exercise: Find the equilibrium point of the Lotka–Volterra model and investigate the oscillations around this equilibrium. Where do the maximum and minimum populations occur? Show that small oscillations around the equilibrium are approximated by the second-order linear oscillator.◆

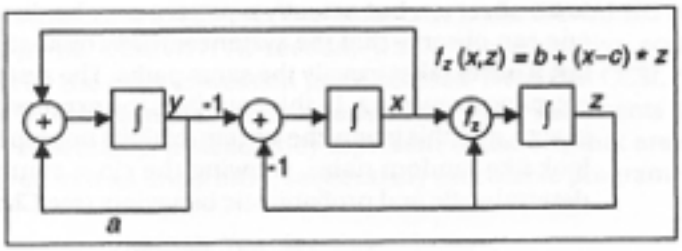

Figure 22 Rössler system.

Although nonlinear systems can show remarkable stability properties, they can also exhibit highly unstable behaviors that appear chaotic. Indeed, the term *chaos* is used by mathematicians for the patterns underlying such "strange" behaviors [1]. Chaotic behavior can come into existence by simple nonlinear feedback. The so-called Rössler attractor is a modified linear oscillator in three variables (Fig. 22). Its typical characteristic is that, independently of the initial condition, the system will come neither to rest nor to some periodic behavior (Fig. 23). The behavior appears to be unpredictable

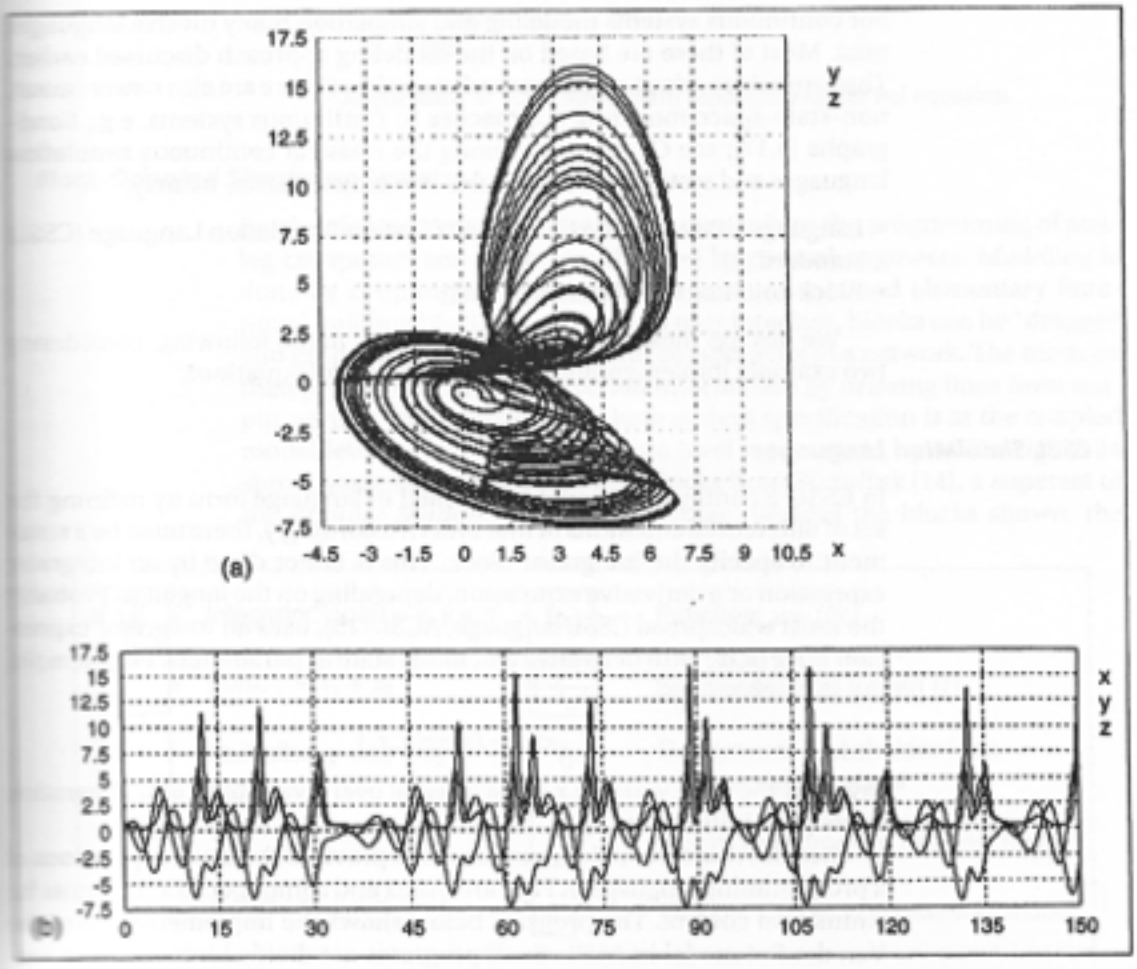

Figure 23 Chaotic behavior of Rössler system: (a) state plane v and z to x, (b) time trajectories.

to an observer, but actually a pattern may be discerned. In the state space one can observe that the system actually oscillates around the zero point, but it never takes exactly the same paths. The system is linear except for the coupling from x to z. In this coupling, the expression $(x - c)$ causes a change in sign, and this is why the system exhibits nonrepetitive fluctuations. These look like random noise, showing the close connection between complex deterministic and probabilistic behaviors (see Chapter 10).

Exercise: Compare cyclic behaviors of the Rössler, Van der Pol, and second-order linear oscillator.◆

Exercise: As with randomness, chaotic behavior can have useful "statistical" properties. For example, the manufacturer of a mobile swimming pool vacuum cleaner certifies that it will cover every inch of the surface every 12 hours. Yet the cleaner never exactly repeats its trajectory through the pool. How might a model exhibiting chaotic behavior justify the manufacturer's claim?◆

3.3.5 Continuous System Simulation Languages and Systems

For continuous systems modeling and simulation many diverse languages exist. Most of these are based on the modeling approach discussed earlier. They are said to adopt a *state-space description*. (There are also newer causal, non-state-space modeling approaches to continuous systems, e.g., Bond-graphs [4,17]; see Chapter 9). Among the classical continuous simulation languages and systems we distinguish two major classes, namely,

- Languages based on the Continuous System Simulation Language (CSSL) standard [2]
- Block-oriented simulation systems

We discuss these two approaches briefly in the following, considering two example implementations of the Van der Pol equations.

CSSL Simulation Languages

In CSSL, a continuous model is specified in language form by defining the set of differential equations of first order. Accordingly, there must be a statement to specify the integrator block. This is either done by an integrator expression or a derivative expression, depending on the language. Probably the most widespread CSSL language, ACSL [15], uses an integrator expression *integ* (d, q_0) with derivative and initial state as parameters. For example, the statement

$$x = integ(v, x_0)$$

says that the state variable x is the integral over a variable v and integration starts from initial value x_0.

Besides this essential statement, ACSL provides the usual expressions of a programming language in Fortran syntax and some special statements for simulation control. The program below shows the implementation of the Van der Pol model in ACSL. Such programs are divided into different sections. In the example, the first section is the initialization section where

some constants are defined. The main part is the DYNAMIC section and within it the DERIVATIVE section, where the derivative equations are defined by the *integ* expression just discussed. Note that CCSL model texts are not programs in the standard sense. The model statements are not executed sequentially, instead, they represent equations that are sorted and translated into an imperative, sequentially executable program.

```
PROGRAM Van der Pol
  INITIAL
    constant
      k = -1, x0 = 1, v0 = 0,
      tf = 20
  END
  DYNAMIC
    DERIVATIVE
      x = integ(v, x0)
      v = integ((1 - x**2)*v - k*x, v0)
    END
    termt (t.ge.tf)
  END
END
```

Algorithm 1 CSSL simulation model of Van der Pol equation.

Block-Oriented Simulation Systems

Block-oriented simulation tools work similarly to the programming of analog computers and are primarily used by control engineers. Modeling is done by coupling together primitive components and elementary functional building blocks. In a graphical user interface, blocks can be "dragged and dropped" into a model to form components in a network. The modeler then provides the interconnection information by drawing lines from output ports to input ports. Thus, here system specification is at the coupled model level as opposed to the state level represented by ACSL. Figure 24 shows some elementary building blocks from *Simulink* [14], a superset of the popular *Matlab* calculation package. Besides the blocks shown, the

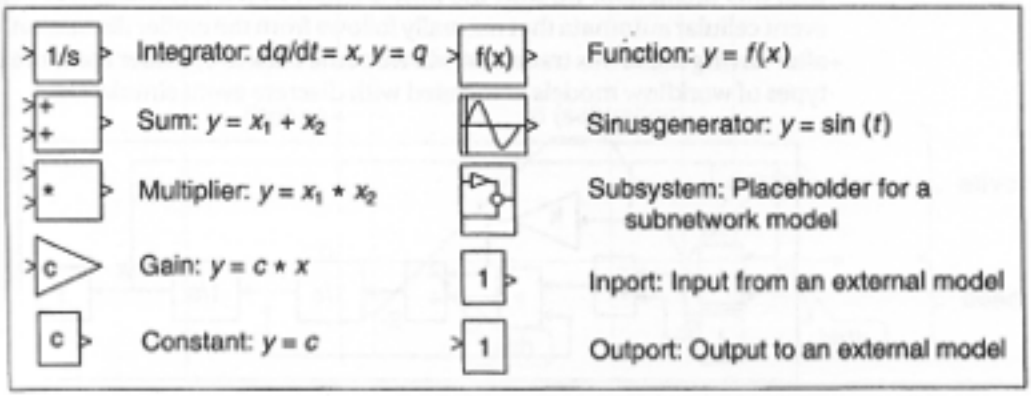

Figure 24 Some building blocks from the Simulink block-oriented simulation package.

Simulink system provides many other blocks that also realize modeling concepts going beyond pure continuous modeling—in particular, building blocks for difference equations and blocks to realize discrete control systems. Our purpose here, however, is to discuss continuous network modeling by means of the Simulink system.

Once more, the most basic block is the *integrator*. It receives one real-valued input and outputs the integral of the input trajectory starting from a particular initial state (a real value which is the first output—initial states are often called initial conditions in the differential equation literature). The integrator therefore realizes a memory element with a single state variable. Besides the integrator, a set of algebraic functions are provided. In Fig. 24 we see building blocks for the sum of two inputs, the multiplication of two inputs, and the gain element that amplifies the input by a constant *c*. There is also a function element that can be programmed in a C-like language. *Sources* are provided to generate test signals (e.g., Constant and Sinus generator).

Simulink allows hierarchical modeling by providing the *subsystem* block as a placeholder for a coupled component and the *inport* and *outport* blocks to model input and output of coupled systems.

Figure 25 shows a block model specification of the Van der Pol equation. The inputs to the integrators are the outputs of a network of algebraic functions. These define the equations for the derivatives of state variables. The network generates the values of state variable *x* as its output.

3.4 Discrete Event Models and Their Simulators

3.4.1 Introduction

We have already seen how a discrete event approach can be taken to obtain more efficient simulation of cellular automata. In this section, we will consider discrete event modeling as a paradigm in its own right. However, the basic motivation remains that discrete event modeling is an attractive formalism because it is intrinsically tuned to the capabilities and limitations of digital computers. Being the "new guy on the block," it still has not achieved the status that differential equations, with a 300, year history, have. But it is likely to play more and more of a role in all kinds of modeling in the future. In keeping with this prediction, we start the introduction with a formulation of discrete event cellular automata that naturally follows from the earlier discussion. Only after having made this transition will we come back to consider the more usual types of workflow models associated with discrete event simulation.

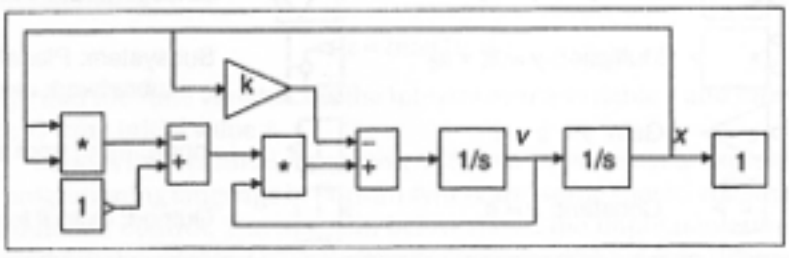

Figure 25 Block model of Van der Pol equations.

3.4.2 Discrete Event Cellular Automata

The original version of the Game of Life assumes that all births and deaths take the same time (equal to a time step). A more accurate representation of a cell's life cycle assumes that birth and death are dependent on a quantity that represents the ability of the cell to fit into its environment, called its *fitness*. A cell attains positive fitness when its neighborhood is supportive, that is, when it has exactly 3 neighbors, and the fitness will diminish rapidly when its environment is hostile. Whenever the fitness then reaches 0, the cell will die. On the other hand, a dead cell will have a negative initial fitness, let's say –2. When the environment is supportive and the fitness crosses the zero level, the cell will be born.

Figure 26 shows an example behavior of such a cell model dependent on its environment (the sum of alive neighbors). The initial sum of alive neighbors being 3 causes the fitness of the dead cell to increase linearly until it crosses zero and a birth occurs. The fitness increases further until it reaches a saturation level (here 6). When the sum of alive neighbors changes from 3 to 4, the environment gets hostile, the fitness will decrease, and the cell will die. The death is accompanied by a discontinuous drop of the fitness level to a minimum fitness of –2.

What is needed to model such a process? One way would be to compute the trajectories of the fitness parameter and see when zero crossings and hence death and birth events occur. This would be the approach of combined discrete event and continuous simulation, which we discuss in detail in Chapter 9. However, this approach is computationally inefficient since we have to compute each continuous trajectory at every point along its path. A much more efficient way is to adopt a pure event-based approach where we concentrate on the interesting events only, namely births and deaths, as well as the changes in the neighborhood. The event-based approach jumps from one interesting event to the next, omitting the uninteresting behavior in between.

The prerequisite of discrete event modeling therefore is to have a means to determine when interesting things happen. Events can be caused by the enviroment, such as the changes of the sum of alive neighbors. The occur-

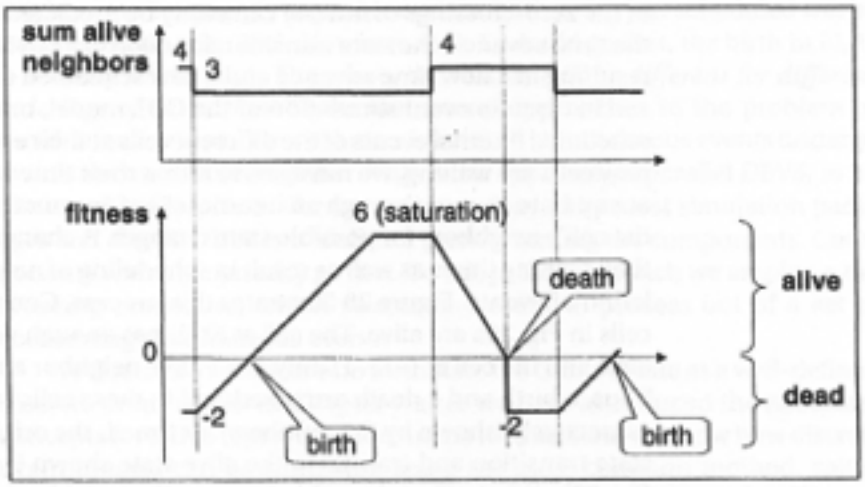

Figure 26 Behavior of GOL model with fitness.

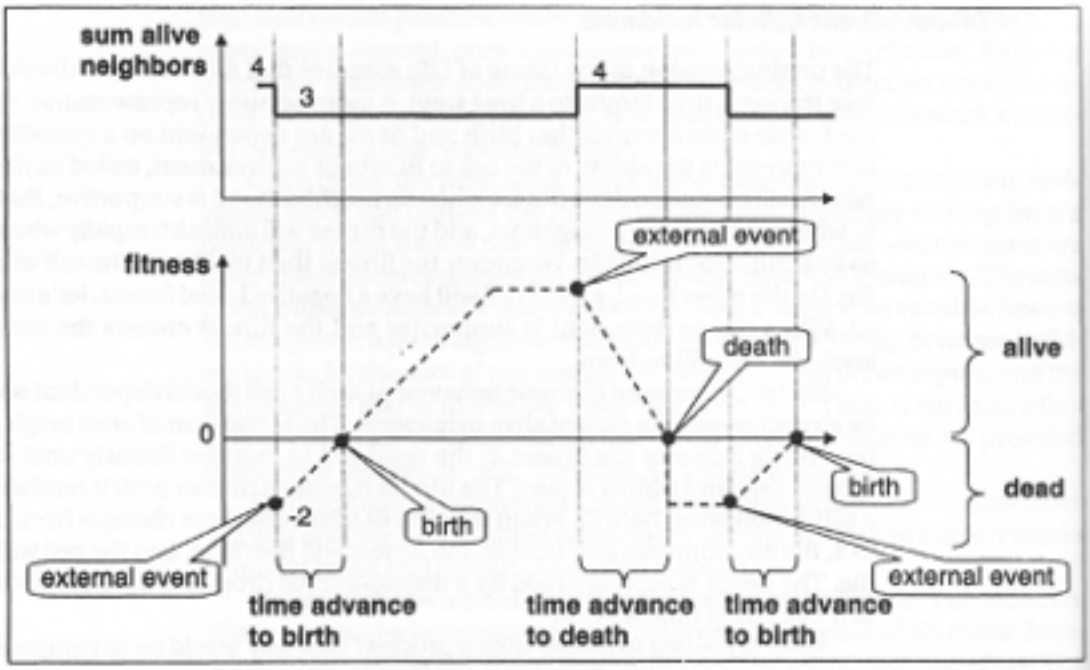

Figure 27 Behavior of the eventistic GOL model.

rences of such *external* events are not under the control of the model component itself. On the other side, the component may schedule events to occur. Those are called *internal* events and the component itself determines their time of occurrence. Given a particular state, e.g., a particular fitness of the cell, a time advance is specified as the time it takes until the next internal event occurs, supposing that no *external event* happens in the meantime. In our event-based GOL model the time advance is used to schedule the times when a birth or a death is supposed to happen (Fig. 27). As fitness changes linearly based on the neighborhood, the times of the events (times of the zero-crossings of fitness) can easily be predicted. Upon expiration of the time advance, the state transition function of a cell is applied, eventually resulting in a new time advance and a new scheduled event.

In discrete event simulation of the GOL model, one has to execute the scheduled internal events of the different cells at their event times. And since now cells are waiting, we must see to it that their time advances. Moreover, at any state change through an internal event we must take care to examine the cell's neighbors for possible state changes. A change in state may affect their waiting times as well as result in scheduling of new events and cancellation of events. Figure 28 illustrates this process. Consider that all shaded cells in Fig. 28a are alive. The cell at (0, 0) has enough neighbors to become alive and the cell at (−1, −1) only has 1 alive neighbor and therefore will die. Thus, a birth and a death are scheduled in these cells at times, say 1 and 2, respectively (shown by the numbers). At time 1, the origin cell undergoes its state transition and transits to the alive state shown by the thunderbolt in Fig. 28b. What effect does this have on the neighboring cells? Figure 28b shows that as a result of the birth, cells at (0, 1), (0, −1), and (1, 1) have three

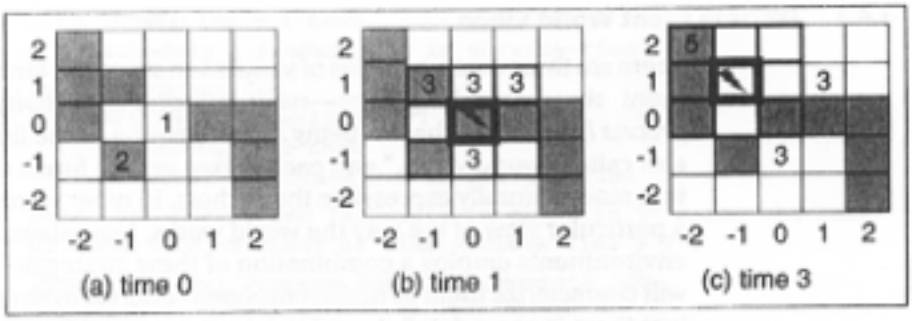

Figure 28 Event processing in Game of Life event model.

neighbors and must be scheduled for births at time 3. Cell (−1, 1) has 4 alive neighbors now and a death will be scheduled for it at time 3. Cell (−1, −1) has had a death scheduled before. However, because of the birth it has two neighbors now, and the conditions for dying do not apply any more. Hence, the death at time 2 must be canceled. We see that *the effect of a state transition may not only be to schedule new events, but also to cancel events that were scheduled in the past.*

Now in Fig. 28 (b), the next earliest scheduled event is at time 3. Therefore, the system can jump from the current time, 1, to the next event time, 3, without considering the times in-between. This illustrates an important *efficiency advantage in discrete event simulation*—during times when no events are scheduled, no components need be scanned. Contrast this with discrete time simulation, in which scanning must be performed at each time step (and involves all cells in the worst case).

The situation at time 3 also illustrates a problem that arises in discrete event simulation—that of *simultaneous events*. From Fig. 28 (b) we see that there are several cells scheduled for time 3, namely a birth at (0, 1), (1, 1) and (0, −1) and a death at (−1, 1). The question is who goes first and what is the result. Note that if (−1, 1) goes first, the condition that (0, 1) has three live neighbors will not apply any more and the birth event just scheduled will be canceled again (Fig. 28c). However, if (0, 1) would go first, the birth in (0, 1) is actually carried out. Therefore, the results *will be different for different orderings of activation.* There are several approaches to the problem of simultaneous events. One solution is to let all simultaneous events undergo their state transitions together. This is the approach of parallel DEVS, to be discussed in Chapter 6. The approach employed by most simulation packages and classic DEVS is to define a priority among the components. Components with high priority go first. To adopt this approach we employ a *tie-breaking* procedure, which selects one event to process out of a set of contending simultaneous events.

We'll return to show how to formulate the Game of Life as a well-defined discrete event model in Chapter 7 after we have introduced the necessary concepts for DEVS representation. In the next section, we show how discrete event models can be simulated using the most common method, called event scheduling. This method can also be used to simulate the Game of Life, and we'll leave that as an exercise at the end of the section.

3.4.3 Discrete Event World Views

There are three common types of simulation strategies employed in discrete event simulation languages—*event scheduling*, *activity scanning*, and *process interaction*, the last being a combination of the first two. These are also called "world views," and each makes certain forms of model description more naturally expressible than others. In other words, each is best for a particular view of the way the world works. Nowadays, most simulation environments employ a combination of these strategies. In Chapter 7, we will characterize them as multicomponent discrete event systems and analyze them in some detail. As an introduction to workflow modeling, here we will now discuss only the simplest strategy, event scheduling.

Event Scheduling World View

Event oriented models *preschedule* all events—there is no provision for conditional events that can be activated by tests on the global state. Scheduling an event is straightforward when all the conditions necessary for its occurrence can be known in advance. However, this is not always the case, as we have seen in the Game of Life. Nevertheless, understanding pure event scheduling in the following simple example of workflow modeling will help to understand how to employ more complex world views later.

Figure 29 depicts a discrete event model with three components. The generator *Gen* generates jobs for processing by a server component *SS*. Jobs are queued in a buffer called *Queue* when the server is busy. Component *Gen* continually reschedules a job output with a time advance equal to *inter-gen-time*. If the server is idle when a job comes in, then it is activated immediately to begin processing the job. Otherwise, the number of waiting jobs in the queue is increased by one. When processing starts, the server is scheduled to complete work in a time that represents the job's *service-time*. When this time expires, the server starts to process the next job in the queue, if there is one. If not, it *passivates* in phase *idle*. To passivate means to set your time advance to infinity. The queue is a *passive* component (its time advance is always infinity) with a variable to store the number of jobs waiting currently.

Two event routines and a tie-breaking function express the interaction between these components:

```
Event: Generate_Job
  nr-waiting = nr-waiting + 1
  schedule a Generate_Job in inter-gen-time
```

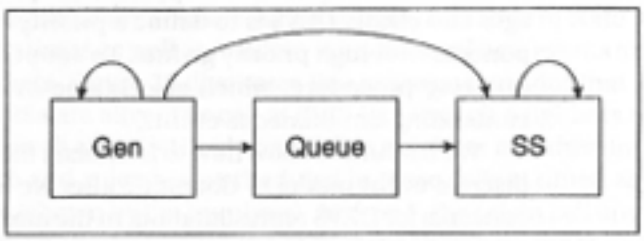

Figure 29 Generator, queue, and server event scheduling model. The arrows indicate the effects of the event routines.

```
    if nr-waiting = 1 then
        schedule a Process_Job in service-time
Event: Process_Job
  nr-waiting = nr-waiting - 1
  if nr-waiting > 0 then
        schedule a Process_Job in service-time
Note that the phrase "schedule an event in T" means the same
as "schedule an event at time = clock time + T".
Break-Ties by: Process_Job then Generate_Job
```

As shown in Fig. 30, the event scheduling simulation algorithm employs a *list of events* that are ordered by increasing scheduling times. The event with the earliest scheduled time (*e*1 in Fig. 30) is removed from the list and the clock is advanced to the time of this imminent event (3). The routine associated with the imminent event is executed. A tie-breaking procedure is employed if there is more than one such imminent events (recall the select function). Execution of the event routine may cause new events to be added in the proper place on the list. For example, *e*4 is scheduled at time 6. Also, existing events may be rescheduled or even canceled (as we have seen).

The next cycle now begins with the clock advance to the earliest scheduled time (5), and so on. Each time an event routine is executed, one or more state variables may get new values.

Let's see how this works in our simple workflow example. Initially, the clock is set to 0 and a *Generate_Job* event is placed on the list with time 0. The state of the system is represented by *nr-waiting*, which is initially 0. Since this event is clearly imminent, it is executed. This causes two events to be scheduled: a *Generate_Job* at time *inter-gen-time* and (since *nr-waiting* becomes 1) a *Process_Job* at time *service-time*. What happens next depends on which of the scheduled times is earliest. Let's assume that *inter-*

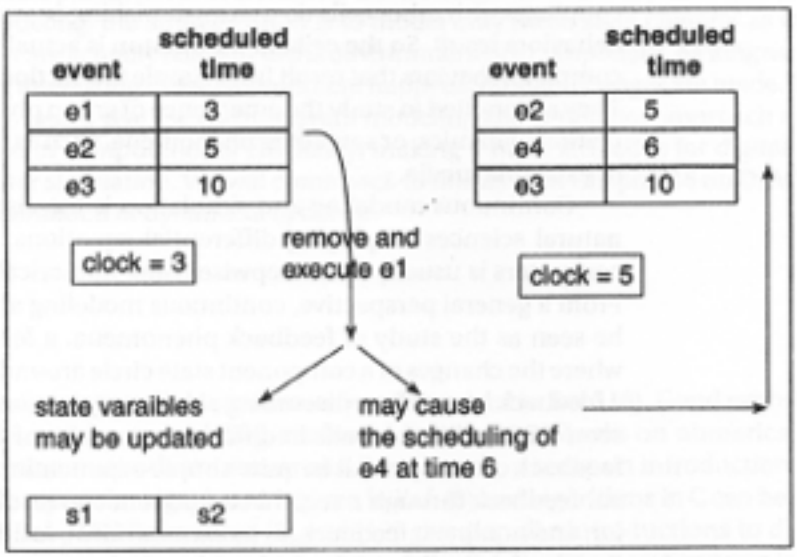

Figure 30 Event list scheduling.

gen-time is smaller than *service-time*. Then time is advanced to *inter-gen-time* and *Generate_Job* is executed. However, now only a new *Generate_Job* is scheduled — since the *nr-waiting* is 2, the *Process_Job* is not scheduled. Let's suppose that *service-time* < 2 * *inter-gen-time* (the next event time for *Generate_Job*). Then the next event is a *Process_Job* and simulation continues in this manner.

Exercise: Hand execute the simulation algorithm for several cases. For example, (*1*) *inter-gen-time* > *service-time*, (*2*) *inter-gen-time* = *service-time*, (*3*) *inter-gen-time* < *service-time* < 2 * *inter-gen-time*, *etc.*◆

Exercise: Use the event scheduling approach to simulate the eventistic Game of Life introduced earlier in the section.◆

3.5 Summary

In this chapter we have discussed the fundamental modeling formalisms that we will introduce in depth in the second part of the book. Here we gained some insight into the intrinsic nature of the different modeling approaches. But the presentation can give us also some insight into the nature of dynamical systems in general. Let us summarize the modeling approaches in this sense.

The discrete time modeling approach, which subsumes the popular finite state automaton formalism as well as the difference equation formalism, stands out through its simple simulation algorithm. It adopts a stepwise execution mode where all the components states are updated based on the state of the previous time step and the inputs.

The simplest form of this model type is the cellular automaton. In a cellular automaton a real system is reduced to basic coupling and communication between components. A cell on its own is quite boring. Only through couplings with other cells do quite astonishing complex and unpredictable behaviors result. So the cellular automaton is actually a model to study the complex behaviors that result from simple interactions among components. They are applied to study the emergence of group phenomena, such as population dynamics, or spreading phenomena, such as forest fires, avalanches, or excitable media.

Continuous modeling and simulation is the classical approach of the natural sciences employing differential equations. Simulation on digital computers is usually done stepwise using numerical integration methods. From a general perspective, continuous modeling and simulation also can be seen as the study of feedback phenomena, a form of system coupling where the changes of a component state circle around to affect itself through a feedback loop. When discussing elementary continuous systems, we have seen that feedback results in different categories of behavior. Direct linear feedback can only result in quite simple exponential behavior; indirect linear feedback through a second component can result in oscillatory behavior; and nonlinear feedback, in its extreme form, results in chaotic behavior. Positive feedback in real world systems is the cause of exponential growth, which sometimes needs to be avoided. Negative feedback loops, on the

other hand, tend to be stabilizing. In any case, the study of feedback lies at the heart of the study of dynamical systems.

Finally, the event-based version of the Game of Life model gave us insight into the intrinsic nature of discrete event modeling and simulation. We have seen that considering only the interesting points in time—the events—is the basis of this modeling approach. To be able to find and schedule the events in time is the prerequisite for applying discrete event simulation. A local event is scheduled by specifying a time advance value based on the current state of a component.

In the GOL model, defining the event times was quite straightforward. Since the fitness parameter increases and decreases linearly, it is simple to compute the time when a zero crossing will occur—that is, when an event occurs. However, in general this will not always be feasible, and several different approaches to tackle this problem are known. A very common approach, which is heavily employed in workflow modeling, is to use stochastic methods and generate the time advance values from a random distribution. Another approach is to measure the time spans for different states and keep them in a table for later look-up. These approaches are discussed in Chapter 16. If none of these methods is appropriate, one has the possibility to step back to the more expensive combined simulation approach where the continuous trajectories are actually computed by continuous simulation and, additionally, events have to be detected and executed in between. This combined continuous/discrete event modeling approach is the topic of Chapter 9.

The GOL event model also showed us two important advantages of discrete event modeling and simulation compared to the traditional approaches. First, only those times, and those components, have to be considered where events actually occur. Second, only those state changes must be reported to coupled components that are actually relevant for their behavior. For example in the GOL model, only changes in the alive state but not changes in the fitness parameter are relevant for a cell's neighbors. Actually, in discrete event modeling, the strategy often is to model only those state changes as events that are actually relevant to the environment of a component. As long as nothing interesting to the environment happens, no state changes are made. These two issues give the discrete event modeling and simulation approach a great lead in computational efficiency, making it most attractive for digital computer simulation. We will come back to this issue in Chapter 16 on DEVS representation of dynamical systems.

3.6 Sources

Research on cellular automata is summarized in Ref. [18]. Good expositions of continuous modeling are found in Refs. [5,9] Books on numerical integration methods are numerous. For example, a compact introduction to the most popular algorithms is given in Ref. [6] and algorithms in C can be found in Ref. [16]. References [3] and [12] give popular introductions to discrete event simulation. A control-oriented view of discrete event systems is given in Ref. [8].

1. Alligood, K. T., T. D. Sauer, and J. A. Yorke, *Chaos: An Introduction to Dynamical Systems*. Springer Verlag, 1996.
2. Augustin, D. C., M. S. Fineberg, B. B. Johnson, R. N. Linebarger, F. J. Sansom, and J. C. Strauss, "The Sci Continuous System Simulation Language (CSSL)." *Simulation* 9, pp. 281–303 (1967).
3. Banks, J., J. Carson, and B. L. Nelson, *Discrete-Event System Simulation*. Prentice Hall, 1995.
4. Blundell, A., *Bond Graphs for Modelling Engineering Systems*. Ellis Horwood Publishers, Chichester, UK, 1982.
5. Bossel, H., *Modeling and Simulation*. Ak Peters, Ltd., 1994.
6. Burden, R. L., and J. D. Faires, *Numerical Analysis*, 4th ed. PWS-KENT Publishing Company, 1989.
7. Burks, A. W., U. Illinois Press, Urbana, IL, 1970. Essays on Cellular Automata.
8. Cassandras, C. G., *Discrete Event Systems: Modeling and Performance Analysis*. Richard Irwin, New York, 1993.
9. Cellier, F. E., *Continuous System Modeling*. Springer-Verlag, New York, 1991.
10. Gardner, M., "The Fantastic Combinations of John Conway's New Solitaire Game of Life." *Scientific American* 23(4), 120–123 (1970).
11. Gould, H. and J. Tobochnik, *An Introduction to Computer Simulation Methods, Parts* 1 *and* 2. Addison-Wesley, 1988.
12. Law, A. M. and W. D. Kelton, *Simulation Modeling and Analysis*, 2nd ed. McGraw hill, New York: 1991.
13. Levins, R., and C. J. Puccia, *Qualitative Modeling of Complex Systems: An Introduction to Loop Analysis and Time Averaging*. Harvard Univ. Press, 1986.
14. Mathworks, Inc., *SIMULINK, User's Guide*. Prentice Hall, Englewood Cliffs, NJ, 1996.
15. Edward E. Mitchell, E. E., and J. S. Gauthier, *ACSL: Advanced Continuous Simulation Language*. Mitchell & Gauthier Assoc., Concord, MA, 1986.
16. Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge University Press, 1992.
17. Thoma, J. U. *Simulation by Bondgraphs*. Springer-Verlag, 1990.
18. Wolfram, S., Ed., *Theory and Application of Cellular Automata*. World Scientific, Singapore, 1986.