# 3 Types of Simulation

This chapter presents an overview of the major types of simulation models with simple illustrations. The systems and/or their models can be discrete, continuous, or a combination. The essential difference in modeling is how the simulation time is advanced. Continuous systems modeling methods such as system dynamics always advance time with a constant delta ($dt$), a fixed time slice. Since variable values may change within any time interval in a continuous system, the delta increment is very small and time-dependent variables are recomputed at the end of each time increment. The variable values change continuously with respect to time. In contrast, in discrete modeling, the changing of variable values is normally event-based. State changes occur in discrete systems at aperiodic times depending on the event happenings. The simulation time is advanced from one event to the next in a discrete manner.

All classes of systems may be represented by any of the model types. A discrete model is not always used to represent a discrete system and vice versa. The choice of model depends on the specific objectives of a study.

Simulation software may be oriented for specific model types or allow hybrid approaches. In general, the software tools must implement the following functions:

- Initialization of system variables
- Time clock for flow control
- For continuous systems:
  - Time advancement in equal small increments, with all variables recomputed at each time step
- For discrete systems:
  - Event calendar for timing
  - Time advancement from one event to the next, with no changes in variables between events
  - Event processing routines
- Statistics collection
- Output generation

Additional features beyond the above may include a graphical user interface, animation, a variety of probability distributions for random variates, design of experiments, optimization, interfaces with other software or databases, and more. See Appendix A for some available simulation tools to support continuous, discrete event, agent-based, and hybrid modeling applications.

Models may be deterministic (having no probabilistic variables) or stochastic (including probabilistic variables). Few engineering applications are wholly deterministic. When a model has probabilistic variables, each run can have a different outcome constituting an estimate of the model characteristics. Therefore, many runs must be made to characterize output variables. See Chapter 4 on handling randomness in models, Chapter 5 for stochastic inputs, and Chapter 6 for output analysis of stochastic systems to address uncertainty.

## 3.1 CONTINUOUS

Continuous systems models consist of differential or algebraic equations to solve variables representing the state of the system over time. A continuous model shows how values change as functions of time computed at equidistant small time steps. These changes are typically represented by smooth continuous curves. Continuous simulation models may also be applied to systems that are discrete in real life but where reasonably accurate solutions can be obtained by averaging values of the model variables.

Continuous models perform a numerical integration of differential equations over time. A practical and simple solution is to use Euler's method. It approximates the derivative of the function $y(t)$ by using the forward difference method. It computes the following at each time increment:

$$y(t_{n+1}) = y(t_n) + \frac{dy}{dt}(t_n)\Delta t \tag{3.1}$$

where

$y$ is a function over time
$\frac{dy}{dt}$ is the time derivative of $y$
$n$ is the time index
$\Delta t$ is the time step size to increment $t_n$ to $t_{n+1}$

The Runge-Kutta method uses additional gradients within the time divisions for more accuracy. See Appendix A for simple programs for continuous systems demonstrating Euler's method and the Runge-Kutta method. These programs use integration methods for computing variable values at each time step within a time loop.

System dynamics is the most widely used form of continuous simulation. System dynamics refers to the simulation methodology pioneered by Jay Forrester, which was developed to model complex continuous systems for improving management policies and organizational structures [5] [6]. Improvement comes from model-based understandings.

System dynamics provides a very rich modeling environment. It can incorporate many formulations including equations, graphs, tabular data, or otherwise. Models are formulated using continuous quantities interconnected in loops of information feedback and circular causality. The quantities are expressed as levels (also stocks or accumulations), rates (also called flows), and information links representing the feedback loops.

Levels represent real-world accumulations and serve as the state variables describing a system at any point in time (e.g. the number of cars on a road, number of manufacturing defects, height of water in a dam, work completed, etc.). Rates are the flows over time that affect the levels. See Table 3.1 for a description of model elements.

The system dynamics approach involves the following concepts [24]:

- Defining problems dynamically, in terms of graphs over time
- Striving for an endogenous (caused within) behavioral view of the significant dynamics of a system
- Thinking of all real systems concepts as continuous quantities interconnected in information feedback loops and circular causality
- Identifying independent levels in the system and their inflow and outflow rates
- Formulating a model capable of reproducing the dynamic problem of concern by itself
- Deriving understandings and applicable policy insights from the resulting model
- Ultimately implementing changes resulting from model-based understandings and insights, which was Forrester's overall goal

A major principle of system dynamics modeling is that the dynamic behavior of a system is largely a consequence of its structure. Thus, system behavior can be changed not only by changing input values but by changing the structure of a system. Improvement of a process thus entails an understanding and modification of its structure. The structures of the as-is and to-be processes are represented in models.

The existence of process feedback is another underlying principle. Elements of a system dynamics model can interact through feedback loops, where a change in one variable affects other variables over time, which in turn affects the original variable. Understanding and taking advantage of feedback effects in real systems can provide high leverage.

### 3.1.1 CONSERVED FLOWS AND THE CONTINUOUS VIEW

In the system dynamics worldview, individual entities are not represented as such. Instead they are abstracted into aggregated flows. The units of flow rate are the number of entities flowing per unit of time. These flows are considered material or physical flows that must be conserved within a flow chain.

Information links connect data from auxiliaries or levels to rates or other auxiliaries. Information connections are not conserved flows because nothing is lost or gained in the data transfer.

A physical example of a real-world level/rate system is a water network, such as a set of holding tanks connected with valved pipes. It's easy to visualize the rise and fall of water levels in the tanks as inflow and outflow rates are varied. The amount of water in the system is conserved within all the reservoirs.

The continuous view does not track individual events, rather flowing entities are treated in the aggregate and systems can be described through differential equations.

There is a sort of blurring effect on discrete events. The focus of continuous models is not on specific individuals or events like discrete event approaches (described in Section 3.2), but instead on the patterns of behavior and on modeling average individuals in a population.

### 3.1.2 MODEL ELEMENTS

System dynamics model elements are summarized in Table 3.1. Their standard graphical notations are shown in the next example model. The same notations with similar icons are used consistently in the modeling tools for system dynamics listed in Appendix A.

TABLE 3.1: System Dynamics Model Elements

| **Element** | **Description** |
|---|---|
| Level | A level is an accumulation over time, also called a stock or state variable. It can serve as a storage device for material, energy, or information. Contents move through levels via inflow and outflow rates. Levels represent the state variables in a system and are a function of past accumulation of rates. |
| Source/Sink | Sources and sinks indicate that flows come from or go to somewhere external to the system or process. Their presence signifies that real-world accumulations occur outside the boundary of the modeled system. They represent infinite supplies or repositories that are not specified in the model. |
| Rate | Rates are also called flows; the actions in a system. They effect the changes in levels. Rates may represent decisions or policy statements. Rates are computed as a function of levels, constants, and auxiliaries. |
| Auxiliary | Auxiliaries are converters of input to output, and help elaborate the detail of stock and flow structures. An auxiliary variable must lie in an information link that connects a level to a rate. Auxiliaries often represent score-keeping variables. |

(*continued*)

TABLE 3.1: System Dynamics Model Elements (*continued*)

| **Element** | **Description** |
|---|---|
| Information Link | Information linkages are used to represent information flow (as opposed to material flow). Rates, as control mechanisms, often require connectors from other variables (usually levels or auxiliaries) for decision making. Links can represent closed-path feedback loops between elements. |

Elements can be combined together to form larger infrastructures with associated behaviors. Using common existing infrastructures in models can save a lot of time and headache when reusing them. An infrastructure can be easily modified or enhanced for different modeling purposes. See [18] for a taxonomy of infrastructures with examples that apply to engineering contexts.

### 3.1.3 MATHEMATICAL FORMULATION OF SYSTEM DYNAMICS

This section explains the mathematics of system dynamics modeling for general background, and is not necessary to develop or use system dynamics models. It illustrates the underpinnings of the modeling approach and provides a mathematical framework for it. Knowing it can be helpful in constructing models. An elegant aspect of system dynamics tools is that systems can be described visually to a large degree, and there is no need for the user to explicitly write or compute differential equations. The tools do all numerical integration calculations. Users do, however, need to compose equations for rates and auxiliaries, which can sometimes be described through visual graph relationships describing two-dimensional (x-y) plots.

The mathematical structure of a system dynamics simulation model is a set of coupled, nonlinear, first-order differential equations per Equation 3.2.

$$x'(t) = f(x,p) \tag{3.2}$$

where

$x$ is a vector of levels

$p$ is a set of parameters

$f$ is a nonlinear vector-valued function

State variables are represented by the levels. As simulation time advances, all rates are evaluated and integrated to compute the current levels. Runge-Kutta or Euler's numerical integration methods are normally used. These algorithms are described in standard references on numerical analysis methods or provided in technical documentation with system dynamics modeling tools. Also see the software examples in Appendix B.

Numerical integration in system dynamics determines levels at any time *t* based on their inflow and outflow rates per Equation 3.3, where the *dt* parameter is the chosen time increment.

$$Level(time) = Level(time - dt) + (inflow - outflow) * dt \tag{3.3}$$

Describing the system with high-level equations spares the modeler from integration mechanics. Note that almost all tools also relieve the modeler of constructing the equations; rather, a diagrammatic representation is drawn and the underlying differential equations are automatically produced.

**Example: System Dynamics Model for Resource Allocation**

Engineering management needs to allocate personnel resources based on relative needs among competing tasks. The model in Figure 3.1 contains resource allocation infrastructures to support this strategic decision making. It is implemented with the iThink tool (see Appendix A). The resource allocation model has levels, shown as boxes, for the tasks (in work units) and task resources (number of people). The two primary flows represent two streams of work: each set of work enters a backlog, is performed at a rate illustrated by the valves, and then enters a completed state. Auxiliary variables are designated as circles (e.g. *Task 1 Productivity*), sources and sinks are the clouds, and information links are the single arrows connecting elements.

In each task flow chain, the rate of work accomplished is represented as a generic production process shown in Figure 3.2. This is an example of a well-established infrastructure. The flow rate drains the task backlog level and accumulates into completed work. The production rate of work completion is computed by multiplying the resources (# of people) by their average productivity (tasks per person per time unit).

Tasks with the greatest backlog of work receive proportionally greater resources ("the squeaky wheel gets the grease"). The allocation infrastructure will adjust dynamically as work gets accomplished, backlogs change, and productivity varies. The dynamic behavior is driven by the equations for the continuously adjusted proportion of resources allocated for a task per Figure 3.3. Resources are allocated dynamically based on each workflow's backlog. More resources are allocated to the workflow of tasks having the larger backlog. As work is accomplished, backlogs change and productivity varies.

Productivity serves as a weighting factor to account for differences in the work rates in calculating the backlog effort. Auxiliaries are used to help compute the estimated effort for each task by adjusting for productivity. The full model equations are shown in Figure 3.4. They include the standard graphic icons for system dynamic elements of rates, levels, and auxiliaries. Graphic curves show defined time-varying input functions (also listed with their table values).

This type of resource policy is used frequently when creating project teams, allocating staff, and planning their workloads. With a fixed staff size, proportionally more people are dedicated to the larger tasks. For example, if one task is about twice as large as another, then that team will have roughly double the people. The work

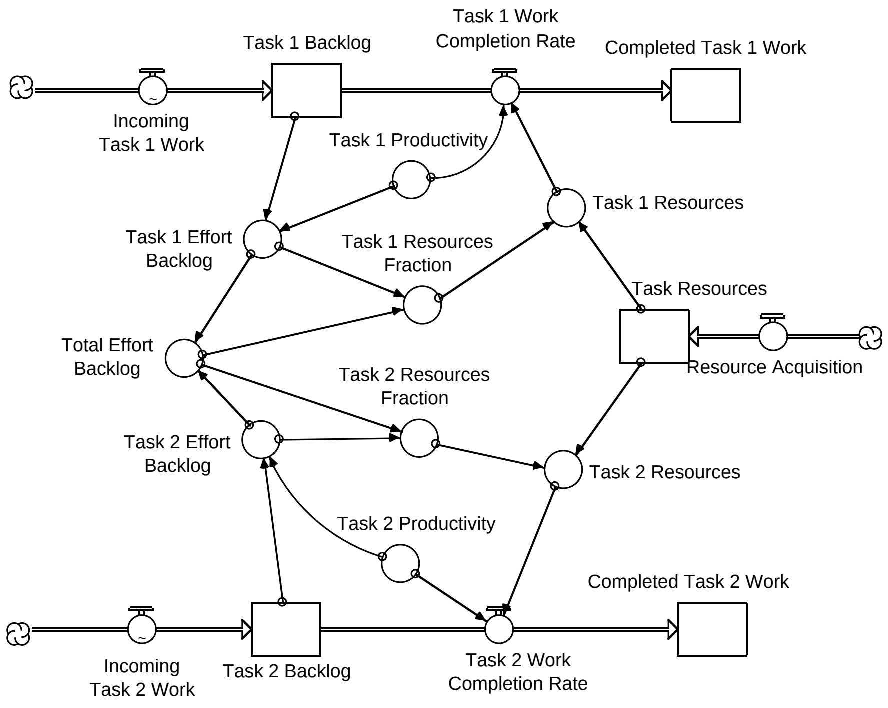
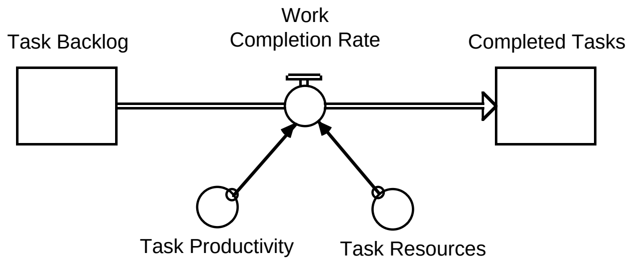


FIGURE 3.1: Resource Allocation Model



Work__Completion_Rate = Task_Resources*Task_Productivity

FIGURE 3.2: Task Production Structure and Rate Equation

backlogs are monitored throughout a project, and people re-allocated as the work ratios change based on the respective estimates to complete. This example models the allocation between two tasks, but it can be easily extended to cover more competing tasks. Many more variations of the resource allocation infrastructure are also possible.

Figure 3.5 shows the results of a simulation run where both tasks start with equal

```
Task_1_Effort__Backlog = Task_1_Backlog/Task_1_Productivity
Task_2_Effort__Backlog = Task_2_Backlog/Task_2_Productivity
Total_Effort__Backlog = Task_1_Effort__Backlog+Task_2_Effort__Backlog
Task_1_Resources__Fraction = Task_1_Effort__Backlog/Total_Effort__Backlog
```

FIGURE 3.3: Task Resource Allocation Equations

```
Completed_Task_1_Work(t) = Completed_Task_1_Work(t - dt) + (Task_1_Work__Completion_Rate) * dt
INIT Completed_Task_1_Work = 0
 INFLOWS:
    Task_1_Work__Completion_Rate = Task_1_Resources*Task_1_Productivity
Completed_Task_2_Work(t) = Completed_Task_2_Work(t - dt) + (Task_2_Work__Completion_Rate) * dt
INIT Completed_Task_2_Work = 0
 INFLOWS:
    Task_2_Work__Completion_Rate = Task_2_Resources*Task_2_Productivity
Task_1_Backlog(t) = Task_1_Backlog(t - dt) + (Incoming__Task_1_Work - Task_1_Work__Completion_Rate)
* dt
INIT Task_1_Backlog = 10
 INFLOWS:
    Incoming__Task_1_Work = GRAPH(time)
    (1.00, 1.00), (2.00, 1.00), (3.00, 1.00), (4.00, 1.00), (5.00, 1.00), (6.00, 1.00), (7.00, 1.00), (8.00, 1.00),
    (9.00, 1.00), (10.0, 1.00), (11.0, 1.00), (12.0, 1.00), (13.0, 1.00)
 OUTFLOWS:
    Task_1_Work__Completion_Rate = Task_1_Resources*Task_1_Productivity
Task_2_Backlog(t) = Task_2_Backlog(t - dt) + (Incoming__Task_2_Work - Task_2_Work__Completion_Rate)
* dt
INIT Task_2_Backlog = 10
 INFLOWS:
    Incoming__Task_2_Work = GRAPH(time)
    (1.00, 1.00), (2.00, 1.00), (3.00, 1.00), (4.00, 1.00), (5.00, 1.00), (6.00, 2.02), (7.00, 1.98), (8.00, 2.00),
    (9.00, 1.00), (10.0, 1.00), (11.0, 1.00), (12.0, 1.00), (13.0, 1.00)
 OUTFLOWS:
    Task_2_Work__Completion_Rate = Task_2_Resources*Task_2_Productivity
Task_Resources(t) = Task_Resources(t - dt) + (Resource_Acquisition) * dt
INIT Task_Resources = 4
 INFLOWS:
    Resource_Acquisition = 0
Task_1_Effort__Backlog = Task_1_Backlog/Task_1_Productivity
Task_1_Productivity = .5
Task_1_Resources = Task_Resources*Task_1_Resources__Fraction
Task_1_Resources__Fraction = Task_1_Effort__Backlog/Total_Effort__Backlog
Task_2_Effort__Backlog = Task_2_Backlog/Task_2_Productivity
Task_2_Productivity = .5
Task_2_Resources = Task_Resources*Task_2_Resources__Fraction
Task_2_Resources__Fraction = Task_2_Effort__Backlog/Total_Effort__Backlog
Total_Effort__Backlog = Task_1_Effort__Backlog+Task_2_Effort__Backlog
```

FIGURE 3.4: Resource Allocation Model

backlogs and 50% allocations due to equal incoming work rates. At *time* = 6 more incoming work starts streaming into the backlog for Task 2 as seen on the graph. The fractional allocations change, and more resources are then allocated to work off the excess backlog in Task 2. The graph for *Task 2 Resources Fraction* increases at that point and slowly tapers down again after some of the Task 2 work is completed.

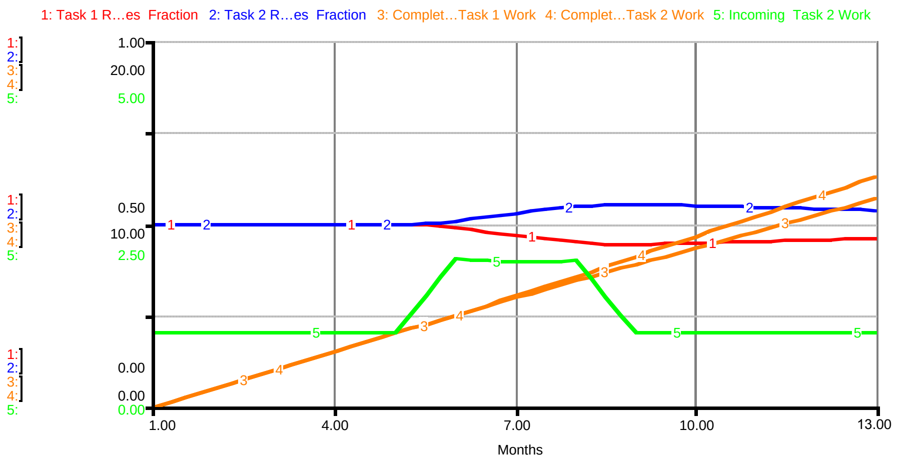


FIGURE 3.5: Resource Allocation Model Output

## 3.2 DISCRETE EVENT

Discrete event models consist of flows of individual entities, characterized by the values of their respective attributes. The characteristics of the entities can change during a simulation run. These changes occur instantaneously as the simulated time lapses. The entities move through a system represented as a network of nodes, perform activities by using resources, and create events that change the state of a system. Standard discrete event model elements are summarized in Table 3.2. They are described in more detail in subsequent sections.

TABLE 3.2: Discrete Event Model Elements

| **Element** | **Description** |
|---|---|
| Create node | Generator of entities to enter a system. |
| Terminate node | Departure point of entities leaving the system. |
| Activity node | Locations where entities are served and consume resources. |
| Entity | An object in a system whose motion may result in an event. |
| Resource | Commodities used by entities as they traverse in a system. |

(*continued*)

TABLE 3.2: Discrete Event Model Elements (*continued*)

| **Element** | **Description** |
|---|---|
| Path | Routes that entities travel between nodes. |
| Batch/Unbatch Nodes | Where entities are combined into groups or uncombined. |
| Information link | Data connections between model elements for logical and mathematical operations. |

## 3.2.1 NEXT EVENT TIME ADVANCE APPROACH

This section describes the discrete event-based approach for a queue/server system and generalizes the introductory example calculations in Section 1.6.1. It illustrates the logic used for time stepping through an event-based simulation, updating of statistics, etc. The next-event time-advance approach for a single-server queueing system will be described using the following notation and illustrated in Figure 3.6. It is the standard procedure used in discrete event modeling. This description (derived from [17]) will step through the computation sequence of a simulation including the generation of random numbers as normally performed by a simulation tool. The nomenclature used is:

$t_i$ = time of arrival of the $i$th customer
$A_i = t_i - t_{i-1}$ = interarrival time between $(i-1)$st and $i$th arrivals of customers
$S_i$ = time that server spends serving $i$th customer
$D_i$ = delay in queue of $i$th customer
$c_i = t_i + D_i + S_i$ = time that $i$th customer completes service and departs
$e_i$ = time of occurrence of $i$th event

The variables are random numbers and we assume that the probability distributions of the interarrival times $A_1$, $A_2$,... and the service times $S_1$, $S_2$,... are known. Their cumulative distribution functions can be used to generate the random variates per the methods described in Chapter 4.

The event times, $e_i$'s, are the values the simulation clock takes on after $e_0 = 0$. At $time = 0$ the server is idle. The time $t_1$ of the first arrival is determined by generating $A_1$ and adding it to 0. The simulation clock is then advanced from $e_0$ to the next event time $e_1$. In Figure 3.6 the curved arrows represent advancing the simulation clock.

The first customer arriving at time $t_1$ finds the server idle and immediately enters service with a delay in queue of $D_1 = 0$. The status of the server is changed from idle to busy. The time $c_1$ when the customer will complete service is computed by generating the random variate $S_1$ and adding it to $t_1$.

The time of the second arrival, $t_2$, is computed as $t_2 = t_1 + A_2$, where $A_2$ is another generated random arrival time. If $t_2 < c_1$ as shown in Figure 3.6, the simulation clock is advanced from $e_1$ to the time of the next event, $e_2 = t_2$. If $c_1$ was less than $t_2$, the clock would be advanced from $e_1$ to $c_1$.

The customer arriving at time $t_2$ finds the server already busy. The number of customers in the queue is increased from 0 to 1 and the time of arrival of this customer is recorded.

Next the time of the third arrival, $t_3$, is computed as $t_3 = t_2 + A_3$. If $c_1 < t_3$ as depicted in the figure, the simulation clock is advanced from $e_2$ to the time of the next event, $e_3 = c_1$, where the first customer completes service and departs. The customer in the queue that arrived at time $t_2$ now begins service. The corresponding delay in queue and service completion times are computed as $D_2 = c_1 - t_2$ and $c_2 = c_1 + S_2$ while the number of customers in the queue is decreased from 1 to 0.

If $t_3 < c_2$ the simulation clock is advanced from $e_3$ to the next event time $e_4 = t3$ as shown in Figure 3.6. If $c_2$ was less than $t_3$, the clock would be advanced from $e_2$ to $c_2$, etc.

With the sequence of events per Figure 3.6, it can be seen that the server is idle between times $c_2$ and $t_3$ before the third customer begins service.

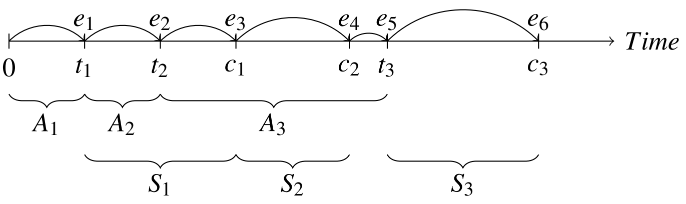


FIGURE 3.6: Next Event Time Advance Approach

## 3.2.2 COMMON PROCESSES AND OBJECTS IN DISCRETE SYSTEMS

Discrete event simulation views systems and processes as interconnected event-based flows of entities through queues and activities. This view corresponds well with intrinsic, measurable, real-world phenomena. Using this view, discrete event simulation represents system behavior using a set of common constructs, described in this section primarily following the concepts and descriptions in [16]. Capabilities to represent these are to be expected of discrete event modeling tools.

In graphical modeling tools these processes may be represented as visual icons called nodes, blocks, modules, or similar names. They may be termed functions or modules in a modeling language. In all cases they represent a model piece or construct with associated behavior logic.

The moving objects in a system whose motion results in events are typically called entities or items. Interconnected nodes represent entity movement routes. The broad similarity of tools is based on movement and accumulation of entities along these connecting paths through the nodes. The modeler logically connects the nodes to represent the entity flows in a system. Model construction entails selection and linking of appropriate model pieces and specifying their parameters.

Knowledge of common processes helps the modeler understand system behavior as well as expected features of modeling tools. The modeler can better identify desirable modeling capabilities for specific problems. Later in this section are graphical and programmed examples of these capabilities with representative and widely used tools in the field. Additional references for more details on discrete event simulation can be found in [17], [2], and [16].

#### Entity Movement

Dynamism in discrete systems is caused by the movement of entities which results in the occurrence of events that change the system state over time. In most systems the entities enter the system as inputs through the system boundary, move between the components within the system, and may leave the system boundary in the form of output from the system. In other cases the system may initially contain some entities that move through the system and possibly leave. Some entities may never leave the system boundary.

#### Entity Creation

To create entities entering a system (e.g. cars arriving for service or manufacturing pieces arriving at a facility) a process for the entity source is required. The simulation module or node type representing this process may be called a *source*, *create*, *generate*, etc. A creation node generates entities with desired timings. It should allow for setting the time of creation of the first entity and the times between entity arrivals as either constant or random variables. For the latter a random variate generator provides samples from probability distribution functions by specifying values of its parameters.

Entities may be assigned names when they are generated. This attribute may be used for entity routing through branches on the basis of the entity type, or for specialized treatment of different entities.

Starting and stopping criteria for creation of entities can be dictated. The creation module should allow for stopping the simulation either after a certain amount of time has elapsed or after a specified number of entities. It is also desirable to allow for the batch creation of entities, i.e., the creation of more than one entity at each arrival time.

One may need to initialize the system with entities already existing at accumulation points. Nodes or modules that correspond to the accumulation points (queues) should provide for initialization of the desired number of entities in each queue.

### Entity Termination

Entities that enter a system may also leave. A provision for entity departure may be called a *terminate*, *sink*, *depart*, *out*, or similar module name in modeling tools to represent the path end for an entity. A simulation may end with some, all, or none of the entities remaining in the system depending on creation and termination parameters. Entity termination may be a simulation stopping criteria. One may specify the number of entity terminations required to end the simulation run.

### Entity Traversal

Entities traverse through paths connecting system components. The paths may have associated delay times or their capacities may be limited. An explicit delay object or node may used to generate a delay corresponding to the travel time of an entity from one node to the other. The delay times may be specified as a constant, a random variable, a user variable, another expression, or be a function of an entity attribute. Path routes can be selectable based on logical conditions, probabilities, or the status of alternative queues on different paths.

Entities traversing a system may also undergo multiplication into more entities, or several entities join to or from a single entity. Carriers can be used to transport batched entities through the network. Grouped items may traverse a carrier entity (e.g. public transit vehicle) and subsequently ungroup. Previously batched entities may become ungrouped, e.g. the unloading of packages from a carrier. Entities that are grouped or assembled together may lose their individual attributes. There may be grouping conditions based on the number of entities or their attributes. An example of different entity types joining would be the assembling of manufactured items from smaller heterogeneous pieces.

### Entity Use of Resources

Resources are commodities used by entities as they traverse a system. They represent servers, operators, machines, etc. and have associated service times. They provide services where entities remain while being processed. Arriving entities must wait until the server is free. Resources (sometimes called facilities) reside at fixed system locations and are made available to entities as needed. Resources usually have a base stock with a fixed level of resource units (e.g. serving capacity).

At event times the entities seize resources, use them with an associated delay, and the resources generally become available again for the next entity. Entity priority schemes are possible, such as special entities that jump to the head of a queue or interrupt another in service. There may also be parallel servers available for entities, each possibly with different service characteristics.

### Entity Accumulation (Queues)

Entities move through the system and may wait at accumulation points. In discrete systems these accumulation points are called *queues* as the entities line up in order.

Queues form when entities need resource facilities that may be busy or otherwise unavailable. A queue represents a buffer before a resource or facility.

Priority schemes may be specified for leaving the queue to be served (e.g. First-In-First-Out (FIFO), Last-In-First-Out (LIFO) and others). Multiple queues may be formed for parallel servers. There may be a queue waiting-area capacity limit constraint, or threshold waiting times may be specified for entities to depart a queue.

Entities may also accumulate and form queues at logical gates or switches on paths where entities require permission to proceed. Rules may be specified before entities enter service, such as a required matching with another entity type. An example is an assembly operation for incoming entities that are synchronized and integrated to form new entities.

### Auxiliary Operations

Auxiliary operations used in discrete event modeling include handling and utilization of variables, manipulation of entities (e.g. transfer, delete, copy), file operations and others. Custom user variables may be specified along with default system variables.

Statistics specification provides for measuring and assessing system performance data. Statistics include both observation-based (e.g. waiting times) and time-based statistics (e.g. utilization) as covered in Sections 1.6.1 and 6.5.1. Capturing of entity event times for arrival, departure, etc., allow for measures such as waiting time or queue length. Server-based statistics include resource utilization measures. For all of these it may be possible to specify only transient or steady-state data collection.

Visual tracing of entities along routes and other animation are additional useful capabilities. These may be helpful to the modeler in debugging and model verification, or external validation and presentation with stakeholders.

**Example: Discrete Processes and Model Objects**

Figure 3.7 shows a graphical ExtendSim [13] model of a single server queue demonstrating basic capabilities for modeling discrete processes with model blocks. Entities are created, they traverse the system, accumulate, use a resource, and terminate. An operation for generating random numbers for event timing is also included.

Entities are generated in the *Create* block and transition to the *Queue* block awaiting service in the *Activity* block. The service time is dictated by a random number sampled from the *Random Number* block and the entities leave the system through the *Exit* block. See further annotations in the figure. Also see Chapter 7 for an applied case study using ExtendSim for modeling discrete processes.

**Example: Car Charging Station Model with SimPy**

Discrete event modeling and simulation will be illustrated in detail for the electric car charging station scenarios. This example is elaborated here and in subsequent chapters to demonstrate basic discrete event model components, key statistics and illustrate the overall modeling process steps. The examples will implement modeling

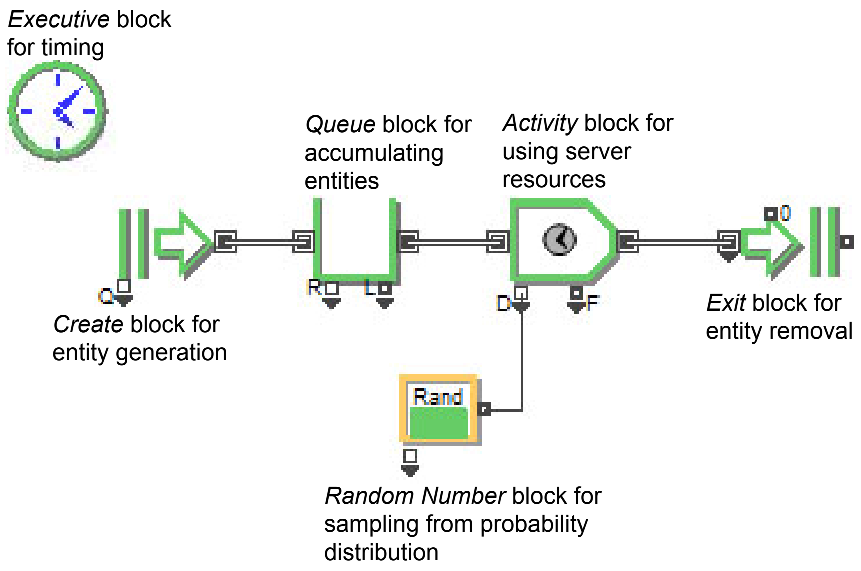

*Executive* block for timing
*Queue* block for accumulating entities
*Activity* block for using server resources
*Create* block for entity generation
*Exit* block for entity removal
Rand
*Random Number* block for sampling from probability distribution
Q
R
L
D
F
0

FIGURE 3.7: ExtendSim Model for Single Server Queue with Common Process Blocks

constructs using the SimPy discrete event simulation library written in the Python language. It is open source and accessible to readers to reproduce these examples, extend the source code provided in this chapter and Appendix B, and use for other applications. See [27] for additional details and downloading of SimPy. An independent technical overview of discrete event simulation using SimPy can also be found in [20].

SimPy is a process-oriented discrete event simulation library, so the behavior of active entities like cars or customers is modeled with *processes*. All processes live in an *environment*. They interact with the environment and with each other via *events*.

Processes are described by simple generators that declare functions that behave like an iterator, i.e., they can be used in a loop. During their lifetime, the processes create events and *yield* them waiting to be triggered. When a process yields an event, the process gets suspended. SimPy resumes the process when the event occurs (gets triggered).

A model for the charging station will require instantiating the SimPy simulation environment, defining an electric car process (generator), and a resource for the charging station. We will illustrate building up the model from the beginning and incrementally add complexity. In the subsequent code examples, any text following the "#" character are embedded comments.

Initializing the SimPy simulation environment can be done as in Figure 3.8. The first statement in the source code imports the SimPy library in order to use its functionality. Creating an instance of the `simpy.Environment` will be needed for a car process and thus will be passed into the car process function. The resources used by car entities will also be attached to it.

```
import simpy

# instantiate execution environment for simulation
environment = simpy.Environment()
```

FIGURE 3.8: Initializing Simulation Environment

The execution environment is a class object with associated methods that will be used. The methods use a dot notation, e.g. `environment.now` refers to the current simulation time.

The electric car objects will first need a resource for the charging station. The resource can be defined per Figure 3.9 with the `simpy.Resource` method specifying the resource environment and characteristics. The resource is parameterized by its capacity (e.g. `capacity=1` refers to a single charging bay). This capacity will be varied later for simulation experiments with different operating scenarios.

```
# charging station resource
charging_station = simpy.Resource(environment,
    capacity=1)
```

FIGURE 3.9: Defining Charging Station Resource

The electric car (event) behavior must also be defined before creating actual car instances. The cars will arrive, use the charging resource, and leave. The car process generator is listed in Figure 3.10 defining the events and is attached to the previously defined charging station resource.

This function in Figure 3.10 generates cars during the simulation with their instrinic process behaviors described in the code for arriving, charging, and departing events. Other actions for statistics collection or output statements can also go in the generator. The car process generator also requires a reference to the named `environment` in order to create new events in it.

The first `yield` statement suspends execution until the arrival time occurs, and then triggers the event for the charging station resource. The resource is requested via the `yield request` statement, and the car will enter service when the charging resource is available. If the server is free at arrival time, then it can start service immediately and the code moves on to the next `yield` for the charging time end event. If the server is busy, the car is automatically queued by the charging resource.

```
# electric car process generator
def electric_car(environment, name, charging_station,
    arrival_time, charging_time):
     # trigger arrival event at charging station
     yield environment.timeout(arrival_time)

     # request charging bay resource
     with charging_station.request() as request:
         yield request
         # charge car battery
         yield environment.timeout(charging_time)
```

FIGURE 3.10: Defining Electric Car Process

When the resource eventually becomes available the car begins service, which ends after the charging time elapses. The car process will pass the control flow back to the simulation once the last `yield` statement is reached.

Next the car arrival times and charging times must be specified before a simulation commences. This first illustration will use the fixed times in Figure 3.11, initially demonstrated in Section 1.6.1. Note that in a typical simulation with randomness, the times are generated within the simulation, and are not a priori constants (random times will be demonstrated next).

```
interarrival_times = [2, 8, 7, 2, 11, 3, 15, 9]
charging_times = [11, 8, 5, 8, 8, 5, 10, 12]
```

FIGURE 3.11: Defining Fixed Event Times

Next a loop to create all the events must be specified, which is shown in Figure 3.12. The code defines a fixed number of car processes to be created. Finally, the simulation is started after the car generation loop by calling the run method `environment.run()`. Note that the simulation duration may be specified in other ways such as a fixed ending time vs. the fixed number of car processes.

Running the program will produce the event output and summary statistics in Figure 3.13 (after adding some output print statements and statistics collection logic listed in Appendix B). The numerical results are identical to the manual calculations in Chapter 1.

Few changes are necessary to model random event times. After importing a random number generation library, the event generation loop simply replaces the fixed times with exponentially distributed random times per Figure 3.14. The provided val-

```
# simulate car processes
for i in range(7):
    arrival_time += interarrival_times[i]
    charging_time = charging_times[i]
    environment.process(electric_car(environment, '
        Car %d' % i, charging_station, arrival_time,
        charging_time))

environment.run()
```

FIGURE 3.12: Event Generation Loop and Simulation Start

ues are reciprocals of the distribution means for the interarrival and charging times respectively (i.e., the interarrival time mean = 6 and charging time mean = 5). The full Python code for these examples with additional print statements, data initializations, and statistics computation is listed in Appendix B.

## 3.3 AGENT BASED

Agent-based modeling uses autonomous decision-making entities called *agents* with rules of behavior that direct their interaction with each other and their *environment*. An agent must possess some degree of autonomy and be distinguishable from its environment. It must perform behaviors or tasks without direct external control reacting to its environment and other agents. Transitions move agents between *states* similar to how rates move entities between levels in continuous models.

Agents modeled may include many types of interacting individuals or groups such people, vehicles, robots, cells, data packets on a network, teams, organizations, and many others. Agents can represent entities that do not have a physical basis but are entities that perform tasks such as gathering information, or modeling the evolution of cooperation. Agent behavior can be reactive, e.g. changing state or taking action based on fixed rules, or adaptive after updating internal logic rules via learning.

Agents interact in environments, which are the spaces in which they behave. The environment may be discrete, continuous, combined discrete/continuous, or characterized by networks. Agent-based modeling can thus be combined with other simulation methods used in engineering sciences. For example, when simulating the interaction with the environment, the environment may be represented by a discrete or continuous field.

Agent-based simulations are a suitable tool to study complex systems with many interacting entities and non-linear interactions among them. Emergent behaviors can result, which are patterns generated by the interactions of the agents, which are often unexpected.

Agent-based modeling is relatively new without an extensive history of engineer-

```
Time  Event
2 Car 0 Arriving at station
2 Car 0 Entering a bay
10 Car 1 Arriving at station
13 Car 0 Charged and leaving
13 Car 1 Entering a bay
17 Car 2 Arriving at station
19 Car 3 Arriving at station
21 Car 1 Charged and leaving
21 Car 2 Entering a bay
26 Car 2 Charged and leaving
26 Car 3 Entering a bay
30 Car 4 Arriving at station
33 Car 5 Arriving at station
34 Car 3 Charged and leaving
34 Car 4 Entering a bay
42 Car 4 Charged and leaving
42 Car 5 Entering a bay
47 Car 5 Charged and leaving
48 Car 6 Arriving at station
48 Car 6 Entering a bay
58 Car 6 Charged and leaving

Summary
# of cars served = 7
Average waiting time: 3.9 minutes
Utilization: 0.948
Average queue length: 0.466
```

FIGURE 3.13: Program Output

ing usage like discrete event and continuous modeling. Its usage is increasing however. Agent-based models have been primarily developed for socio-economic systems simulating the interactions of autonomous agents (both individual or collective entities such as organizations or groups) to assess their effects on the system as a whole.

There are also many scientific and engineering applications to model groups of actors and their interactions based on behavioral rules. Urban systems in civil engineering to study people patterns, communities, vehicles, and traffic patterns are a natural fit. Example applications to-date in civil engineering include water resources [3], civil infrastructure [26], construction management [28], and more. They have been used to model complex healthcare systems and disaster response to simulate the movement of many individual people, and the consequent behavior of the crowd,

```
# simulate car processes
for i in range(7):
     arrival_time += random.expovariate(0.16)
     charging_time = random.expovariate(0.2)
     environment.process(electric_car(environment, '
        Car %d' % i, charging_station, arrival_time,
        charging_time))

environment.run()
```

FIGURE 3.14: Event Generation with Random Times

as they make their way out of the buildings and find transport to medical facilities.

Modeling dynamically interacting rule-based agents is ideal for transportation systems with emergent phenomena, when individual behavior is nonlinear and changes over time with fluctuations like traffic jams. It can be more natural to describe individual behavior activities compared to using processes in discrete event models.

Agent-based modeling is relevant for the engineering of systems of systems. Systems engineers can investigate alternative architectures and gain an understanding of the impact of the behaviors of individual systems on emergent behaviors. Other examples of engineering applications include supply chain optimization, logistics, distributed computing, and organizational and team behavior.

### 3.3.1 AGENT-BASED MODEL ELEMENTS

Agent-based modeling is typically implemented with an object-oriented programming where data and methods (operations) are encapsulated in objects that can manipulate their own data and interact with other objects. The behaviors and interactions of the agents may be formalized by equations, but more generally they may be specified through decision rules, such as if-then statements or logical operations.

Table 3.3 lists typical agent-based model elements. These definitions are consistent with the implementation in [8] and may vary slightly in other modeling tools.

TABLE 3.3: Agent-Based Model Elements

| **Element** | **Description** |
|---|---|
| Population | An interacting group or collection of agents during a simulation described by the type of agents and the population size. |
| Agent | An individual actor in a population governed by its own rules of behavior. |
| State | An on/off switch for an agent. When the state is active, the agent is in that state and vice versa. One or more sets of states may be placed in an agent. |
| Transition | Transitions move agents between states in a model. When a transition is activated, an agent moves from one state to another. They are configured by what triggers the transition. Some different types of triggers may include a timeout, a probability, or a condition based on logical relationships to other agents or events. |
| Action | An action manipulates agents during a simulation. They are defined by what triggers the action and what the action does when triggered. An action can be to move an agent, to change a primitive's value, to add a connection to an agent, or other. |

**Example: Agent-Based Model for Car Behavior**

Figure 3.15 shows an agent-based model for electric cars acting as interacting agents. This is a web-based simulation model using Insight Maker (see Appendix A). Consistent with the elements in Table 3.3, the population of cars is signified by the cloud at the top. Cars are defined as the agents that move between the states of *Waiting*, *Parked*, *Impatient*, and *Driving*.

This model is essentially a queuing process like the previous car charging example, but a variation with respect to the discrete event model is the consideration of cars leaving after waiting too long. Note the car behavior is actually dictated by a human agent inside and could alternatively be termed a driver rather than a vehicle.

Transitions are indicated by arrows in the model for moving between states. These are triggered by the decision to start looking for a charging space, the existence of an empty space, giving up after waiting too long, and completion of charging to

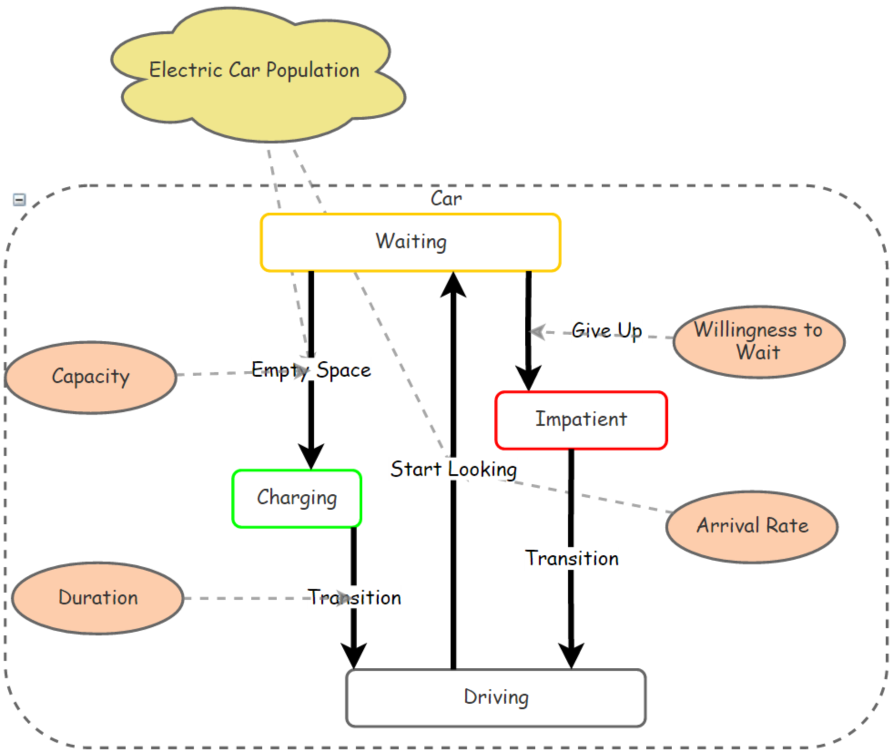


FIGURE 3.15: Example Agent-Based Model for Electric Car Charging

start driving again. The ovals are other model variables used in the logical rules of behavior. Dashed lines represent information links passing values.

A population of cars start off in a *Driving* state. At each cycle, according to a Poisson distribution defined by *Arrival Rate*, some cars transition to *Waiting* as they wait for an empty charging space according to:

```
RandPoisson(Round([Arrival Rate])/Count(FindState([
    Electric Car Population], [Driving]))
```

The number of cars in the *Waiting* state represents a queue. If an empty bay is available for a car, then the state transitions to *Charging* per:

```
[Parking Capacity] > Count(FindState([Electric Car
    Population],[Charging]))
```

The cars remain in a *Charging* state according to a normal distribution and then transition back to *Driving* as they leave, modeled as:

```
RandNormal([Duration], [Duration]/2)
```

The model uses a timeout for the impatience behavior. If a car is in the *Waiting* state for a period longer than *Willingness to Wait*, then the state times out and transitions to *Impatient* and immediately transitions to *Driving* again. Figures 3.16 and 3.17 show a simulation timeline and state transition map for a simulation run.

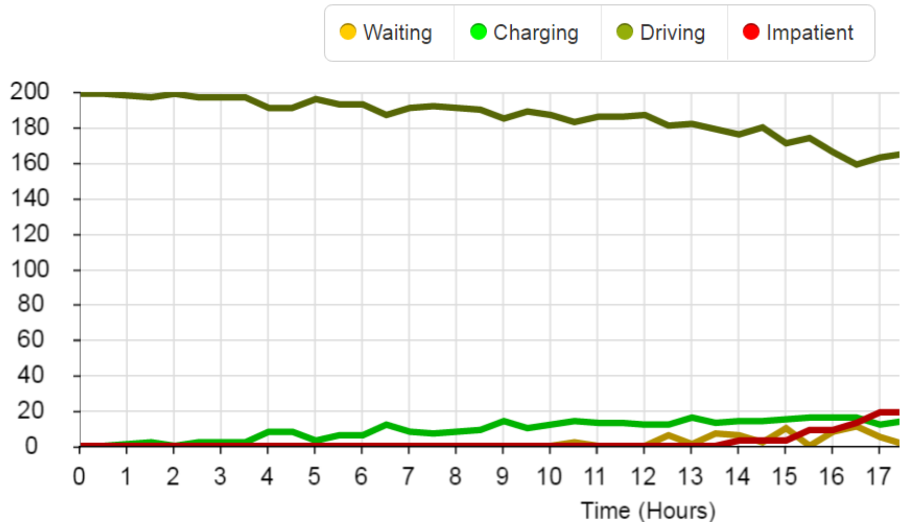


FIGURE 3.16: Simulation Timeline

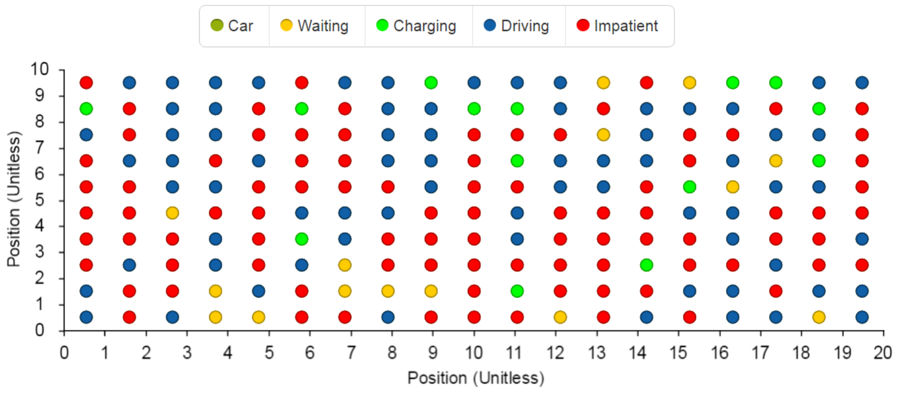


FIGURE 3.17: State Transition Map

## 3.4 SUMMARY

Systems and/or their models can be discrete, continuous, or a combination. All classes of systems may be represented by any of the model types. In continuous models the variables change continuously with respect to time. Time advancement is in equal small increments, with all variables recomputed at each time step. Discrete system models are event based, whereby an event calendar is used for timing. State changes occur in discrete systems at aperiodic times depending on the event times. Time advancement is from one event to the next, with no changes in variables between events.

Before choosing a modeling method for engineering applications, an important question that should be answered is, "What is continuous and what is truly discrete?" Are there discrete aspects that need to be preserved for the purpose of the study or is an aggregate view sufficient?

System dynamics is the most widely used form of continuous simulation. In the continuous view, process entities are represented as aggregated flows over time. The approach does not track individual events; rather, flowing entities are treated in the aggregate and systems can be described through differential equations. System dynamics provides a rich modeling environment incorporating many types of formulations for this.

System dynamics model elements are levels, sources and sinks, rates (flows), information links, and auxiliary variables. Levels represent real-world accumulations and serve as the state variables describing a system at any point in time. Rates are the flows over time that affect the levels. Rates may represent decisions or policy statements. Rates are computed as a function of levels, constants, and auxiliaries. Sources and sinks indicate that flows come from or go to somewhere external to the system. Models are built using conserved flow chains where the levels store the flowing entities.

Discrete event models consist of flows of individual entities, characterized by the values of their respective attributes. The characteristics of the entities can change during a simulation run. These changes occur instantaneously as the simulated time lapses. The entities move through a system represented as a network of nodes, perform activities by using resources, and create events that change the state of a system. The next-event time-advance approach is the standard procedure used for computing in discrete event modeling.

Discrete event simulation represents system behavior using a set of common constructs that correspond well with intrinsic, measurable, real-world phenomena. Standard discrete processes include entity creation, entity movement and traversal, entity use of resources, entity accumulation (queues), and entity termination. In tools they are used to represent the movement and accumulation of entities along paths through system nodes. Modeling tools also provide auxiliary operations and statistics specification for measuring and assessing system performance data.

Agent-based models contain agents that are autonomous decision-making entities with rules of behavior that direct their interaction with each other and their environment. Agents may include many types of interacting individuals or groups. Agents interact in environments, which are the spaces in which they behave. The environment may be discrete, continuous, combined discrete/continuous, or characterized by networks.

Agent-based model elements include agents, their populations, states, transitions, and actions. Individual agents are governed by their own rules of behavior and populations store agents during a simulation. Agents have internal states, and transitions are used to move agents between states in a model. Actions manipulate agents during a simulation and are defined with event triggers.

This chapter has demonstrated simple, illustrative models for continuous, discrete event and agent-based types of modeling and simulation. The reader is encouraged to look further in the references and upcoming chapters, see Appendix B for modeling tools, experiment with the tools and the provided models in this book.