# MCA 504
# Modelling and Simulation

Index

**Subject : System Simulation and Modeling** **Author : Jagat Kumar**
**Paper Code: MCA 504** **Vetter : Dr. Pradeep Bhatia**
**Lesson : System Models and System Simulation**
**Lesson No. : 01**

**Structure**

## 1.0 Objective

The main objective of this module to gain the knowledge about system and its behavior so that a person can transform the physical behavior of a system into a mathematical model that can in turn transform into a efficient algorithm for simulation purpose.

## 1.1 Introduction

Computer simulation is a powerful methodology for design and analysis and complex systems. The overall approach in computer simulation is to represent the dynamic characteristics of a real world system in a computer model. The model is subjected to experiments to obtain predictive information useful in making informed decision making about the characteristics of the real system. Simulations are suitable for problems in which there are no closed-form analytical solutions. Since most dynamic problems in practice can not be represented and solved fully using mathematical equations, computer simulation is a powerful and flexible methodology in complex systems analysis.
Simulations can be classified into continuous and discrete simulations. In continuous simulations, the state variables, i.e., the collection of variables needed to describe the system, change continuously over time and the behavior of the system is typically described by differential equations. Examples of continuous systems include the modeling of thermal or hydraulic systems. Discrete simulations are event-driven where the state variables change at discrete time points. Examples of discrete-event simulations include service industry applications such as queues in a grocery store and manufacturing applications involving material flow analysis. In general we have three different methods as shown in Figure 1 to study a real system

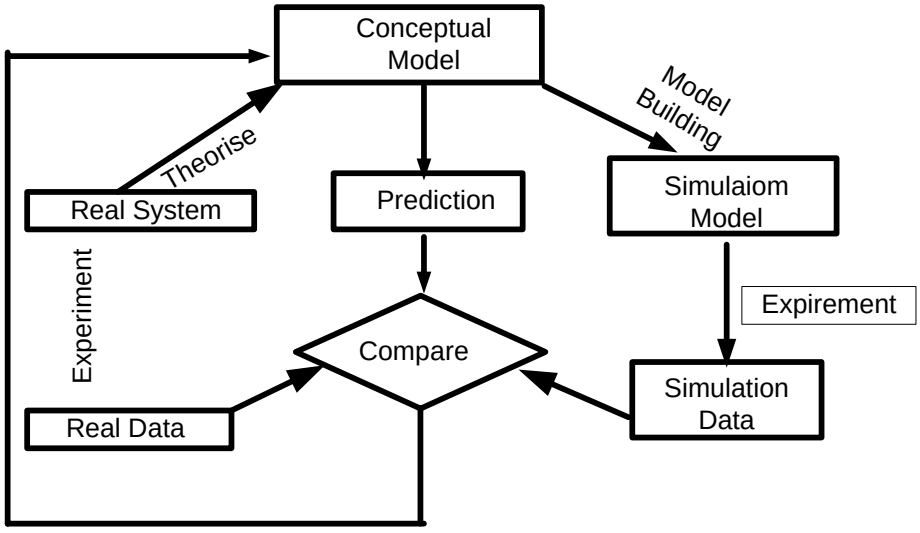



**Figure 1 : Three Methods of Science**

Briefly we can say that Simulation is
• Simulated system imitates operation of actual system over time
• Artificial history of system can be generated and observed
• Internal (perhaps unobservable) behavior of system can be studied
• Time scale can be altered as needed
• Conclusions about actual system characteristics can be inferred
in Figure 2 , actual system (real system) is compared with simulation

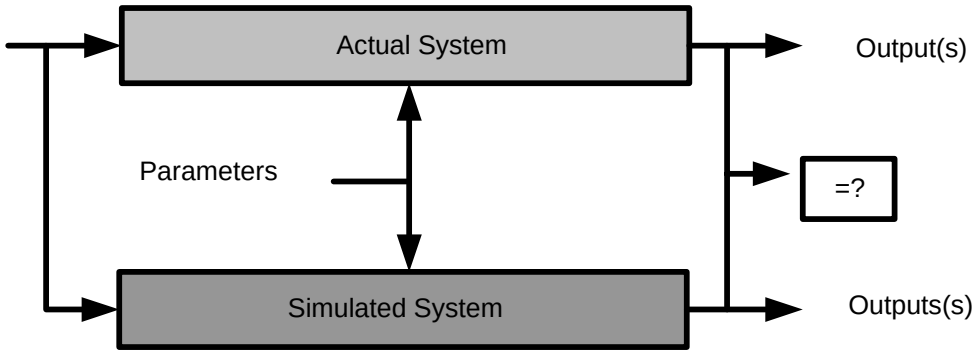



**Figure 2: Simulation vs Actual System**

### 1.1.1 Formal Definition(s)

Simulation can be broadly defined as *a technique for studying real-world dynamical systems by imitating their behavior using a mathematical model of the system implemented on a digital computer*.

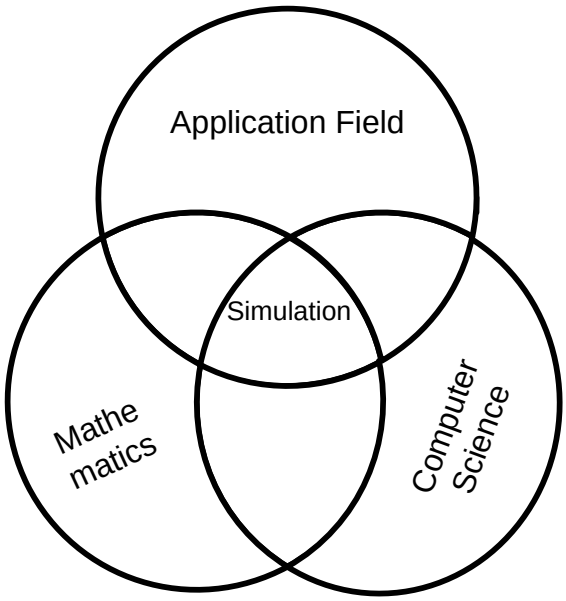



**Figure 3: Simulation is Interdisciplinary**

Simulation can also be viewed as *a numerical technique for solving complicated probability models*, ordinary differential equation and partial differential equation, analogously to the way in which we can use a computer to numerically evaluate the integral of a complicated function. That's why science of simulation is considered as an interdisciplinary subject a shown in Figure 3.

### 1.1.2 A Brief History of Simulation

**1940's:** Monte Carlo method is developed by physicists working on Manhattan project to study neutron scattering. Researchers include John von Neumann, Stanis law Ulan, Edward Teller, Herman Kahn

**1950's:** First special-purpose simulation languages developed (e.g. IMSCRIPT by Harry Markowitz at RAND Institute)
**1970's:** Research initiated on mathematical foundations of simulation
**1980's:** PC-based simulation software developed, graphical user interfaces, object-oriented programming
**1990's:** Web-based simulation, fancy animated graphics, simulation-based optimization, Markov-chain Monte Carlo methods Simulation has become ever more prominent as a method for studying complex systems in which uncertainty is present. In various surveys, simulation has been found to be the most frequently used tool of Operation Research practitioners. Simulation is an interdisciplinary subject, using ideas and techniques from Statistics, Probability, Number Theory, and Computer Science.

### 1.1.3 Application Areas of Simulation

- Manufacturing
- Computer Systems
- E-business/workflow systems
- Finance
- Telecommunications
- Transportation
- Military

1.1.4 Advantages and Disadvantages of Simulation

**Advantages:** Simulation arbitrary model complexity, circumvents analytically intractable models, facilitates what-if and sensitivity analyses, building a model can lead to system improvements and greater understanding can be used to verify analytic solutions

**Disadvantages:** Simulation provides only estimates of solution, only solves one parameter at a time, can take a large amount of development and/or computer time ("simulation as a last resort"). Don't use computer simulation if a common-sense or analytical solution is available, or if resources are insufficient, or if simulation costs outweigh benefits.

### 1.1.5 Difficulties of Simulation

- Provides only individual, not general solutions
- Manpower and time-consuming
- Computing memory and time-intensive
- Difficult so experts are required
- Hard to interpret results
- Expensive

### 1.1.6 When to Use Simulation?

- Study internals of a complex system e.g. biological system
- Optimise an existing design e.g. routing algorithms, assembly line
- Examine effect of environmental changes e.g. weather forecasting
- System is dangerous or destructive e.g. atom bomb, atomic reactor, missile launching
- Study importance of variables
- Verify analytic solutions (theories)
- Test new designs or policies
- Impossible to observe/influence/build the system
- When it allows inspection of system internals that might not otherwise be observable
- Observation of the simulation gives insights into system behavior
- System parameters can be adjusted in the simulation model allowing assessment of their sensitivity (scale of impact on overall system behavior)
- Simulation verifies analysis of a complex system, or can be used as a teaching tool to provide insight into analytical techniques
- A simulator can be used for instruction, avoiding tying up or damaging an expensive, actual system (e.g., a flight simulation vs. use of multimillion dollar aircraft)

## 1.2 Modelling Concepts

There are several concepts underlying simulation. These include system and model, events, system state variables, entities and attributes, list processing, activities and delays, and finally the definition of discrete-event simulation.
The process of making and testing hypotheses about models and then revising designs or theories has its foundation in the experimental sciences. Similarly, computational scientists use **modeling** to analyze complex, real-world problems in order to predict what might happen with some course of action. For example, Dr. Jerrold Marsden, a computational physicist at CalTech, models space mission trajectory design (Marsden). Dr. Julianne Collins, a genetic epidemiologist (statistical genetics) at the Greenwood Genetics Center, runs genetic analysis programs and analyzes epidemiological studies using the Statistical Analysis Software (SAS) (Greenwood Genetics Center). Some of the projects on which she has worked involve analyzing data from a genome scan of

Alzheimer's disease, performing linkage analyses of X-linked mental retardation families, determining the recurrence risk in nonsyndromic mental retardation, analyzing folic acid levels from a nutritional survey of Honduran women, and researching new methods to detect genes or risk factors involved in autism. Scientists in areas such as cognitive psychology and social psychology at the Human-Technology Interaction Center of The University of Oklahoma perform research on the interaction of people with modern technologies (Human-Technology Interaction Center). Some of the studies involve "strategic planning in air traffic control" and "designing interfaces for effective information retrieval from collections of multimedia." Buried land mines are a serious danger in many areas of the world (Weldon et al. 2001). Scientists are using a combination of mathematics, signal processing, and scientific visualization to model, image, and discover land mines. Lourdes Esteva, Cristobal Vargas, and Jorge Velasco-Hernandez have modeled the oscillating patterns of the disease dengue fever, for which an estimated 50 to 100 million cases occur globally each year (Esteva and Vargas 1999).

### 1.2.1 System, Model and Events

**A model is a representation of an actual system** (Figure 4) and Figure 5 presents modelling and simulation concepts as introduced by Zeigler [2].

- A model is an abstraction of the real system
- Simplifying assumptions are used to capture (only) important behaviors
- Linearization, time-bound behaviors, etc., may make analysis tractable

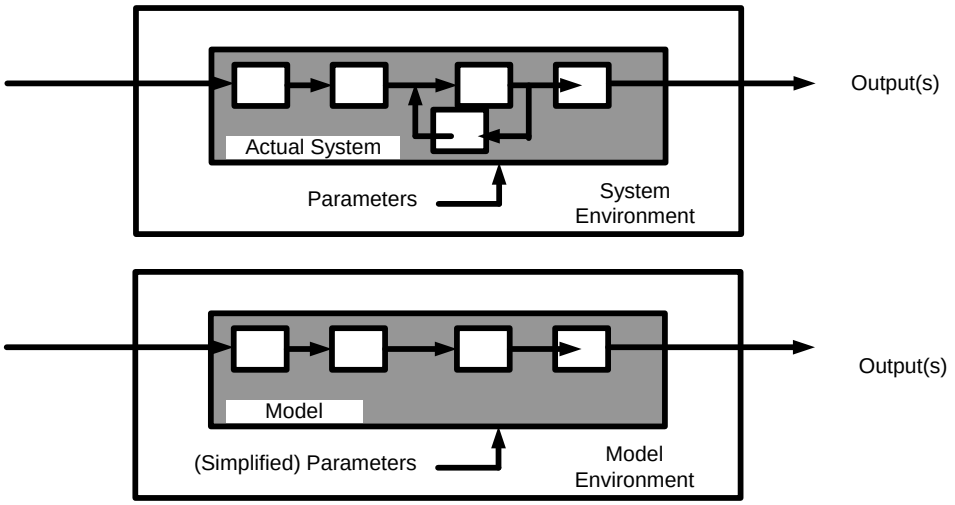



**Figure 4 : Pictorial Representation of System Model**

**Formally we can define, Modeling** is the application of methods to analyze complex, real-world problems in order to make predictions about what might happen with various actions.

**Object** is some entity in the Real World. Such an object can exhibit widely varying behavior depending on the context in which it is studied, as well as the aspects of its behavior which are under study.

**Base Model** is a hypothetical, abstract representation of the object's properties, in particular, its behavior, which is valid in *all* possible contexts, and describes all the object's facets. A base model is hypothetical as we will never in practice be able to construct/represent such a total model. The question whether a base model exists at all is a philosophical one.

**System** is a well defined object in the Real World under specific conditions, only considering specific aspects of its structure and behaviour.

**Experimental Frame** When one studies a system in the real world, the experimental frame (EF) describes experimental conditions (context), aspects, within which that system and corresponding models will be used. As such, the Experimental Frame reflects the *objectives* of the experimenter who performs experiments on a real system or, through simulation, on a model.

Immediately, there is a concern about the limits or boundaries of the model that supposedly represent the system. The model should be complex enough to answer the questions raised, but not too complex. Consider an event as an occurrence that changes the state of the system. In the example, events include the arrival of a customer for service at the bank, the beginning of service for a customer, and the completion of a service. There are both internal and external events, also called endogenous and exogenous events, respectively. For example, an endogenous event in the example is the beginning of service of the customer since that is within the system being simulated. An exogenous event is the arrival of a customer for service since that occurrence is outside of the simulation. However, the arrival of a customer for service impinges on the system, and must be taken into consideration.

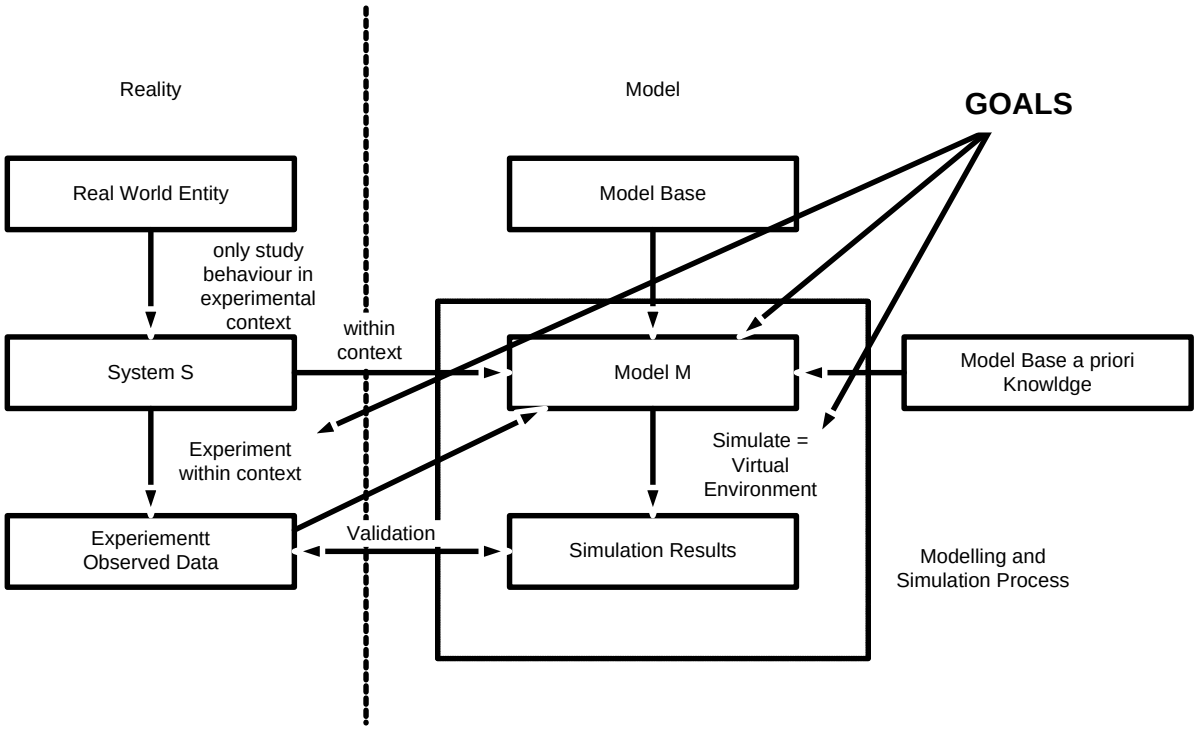



**Figure 5: Modeling and Simulation**

Discrete-event simulation models are contrasted with other types of models such as mathematical models, descriptive models, statistical models, and input-output models. A discrete-event model attempts to represent the components of a system and their interactions to such an extent that the objectives of the study are met. Most mathematical, statistical, and input output models represent a system's inputs and outputs explicitly, but represent the internals of the model with mathematical or statistical relationships. An example is the mathematical model from physics,

Force = Mass × Acceleration

based on theory. Discrete-event simulation models include a detailed representation of the actual internals.
Discrete-event models are dynamic, i.e., the passage of time plays a crucial role. Most mathematical and statistical models are static in that they represent a system at a fixed point of time. Consider the annual budget of a firm. This budget resides in a spreadsheet. Changes can be made in the budget and the spreadsheet can be recalculated, but the passage of time is usually not a critical issue. Further comments will be made about discrete-event models after several additional concepts are presented.

**Models have Many Uses, Typically**

- To understand the behaviour of an existing system (why does my network performance die when more than 10 people are at work?)

- To predict the effect of changes or upgrades to the system (will spending 100,000 on a new switch cure the problem?)
- To study new or imaginary systems (let's bin the Ethernet and design our own scalable custom routing network)

### 1.2.2 System State Variables

The system state variables are the collection of all information needed to define what is happening within the system to a sufficient level (i.e., to attain the desired output) at a given point in time. The determination of system state variables is a function of the purposes of the investigation, so what may be the system state variables in one case may not be the same in another case even though the physical system is the same.
Determining the system state variables is as much an art as a science. However, during the modeling process, any omissions will readily come to light. (And, on the other hand, unnecessary state variables may be eliminated.) Having defined system state variables, a contrast can be made between discrete-event models and continuous models based on the variables needed to track the system state. The system state variables in a discrete-event model remain constant over intervals of time and change value only at certain well-defined points called event times. Continuous models have system state variables defined by differential or difference equations giving rise to variables that may change continuously over time.
Some models are mixed discrete-event and continuous. There are also continuous models that are treated as discrete-event models after some re-interpretation of system state variables, and vice versa.

#### 1.2.2.1 Entities and Attributes

An entity represents an object that requires explicit definition. An entity can be dynamic in that it "moves" through the system, or it can be static in that it serves other entities. In the example, the customer is a dynamic entity, whereas the bank teller is a static entity. An entity may have attributes that pertain to that entity alone. Thus, attributes should be considered as local values. In the example, an attribute of the entity could be the time of arrival. Attributes of interest in one investigation may not be of interest in another investigation. Thus, if red parts and blue parts are being manufactured, the color could be an attribute. However, if the time in the system for all parts is of concern, the attribute of color may not be of importance. From this example, it can be seen that many entities can have the same attribute or attributes (i.e., more than one part may have the attribute "red").

#### 1.2.2.2 Resources

A resource is an entity that provides service to dynamic entities. The resource can serve one or more than one dynamic entity at the same time, i.e., operate as a parallel server. A dynamic entity can request one or more units of a resource. If denied, the requesting entity joins a queue, or takes some other action (i.e. diverted to another resource, ejected from the system). (Other terms for queues include files, chains, buffers, and waiting

lines.) If permitted to capture the resource, the entity remains for a time, then releases the resource.
There are many possible states of the resource. Minimally, these states are idle and busy. But other possibilities exist including failed, blocked, or starved.

#### 1.2.2.3 List Processing

Entities are managed by allocating them to resources that provide service, by attaching them to event notices thereby suspending their activity into the future, or by placing them into an ordered list. Lists are used to represent queues. Lists are often processed according to FIFO (first-in first-out), but there are many other possibilities. For example, the list could be processed by LIFO (last-in-first out),according to the value of an attribute, or randomly, to mention a few. An example where the value of an attribute may be important is in SPT (shortest process time) scheduling. In this case, the processing time may be stored as an attribute of each entity. The entities are ordered according to the value of that attribute with the lowest value at the head or front of the queue.

#### 1.2.2.4 Activities and Delays

An activity is duration of time whose duration is known prior to commencement of the activity. Thus, when the duration begins, its end can be scheduled. The duration can be a constant, a random value from a statistical distribution, the result of an equation, input from a file, or computed based on the event state. For example, a service time may be a constant 10 minutes for each entity, it may be a random value from an exponential distribution with a mean of 10 minutes, it could be 0.9 times a constant value from clock time 0 to clock time 4 hours, and 1.1 times the standard value after clock time 4 hours, or it could be 10 minutes when the preceding queue contains at most four entities and 8 minutes when there are five or more in the preceding queue. A delay is an indefinite duration that is caused by some combination of system conditions. When an entity joins a queue for a resource, the time that it will remain in the queue may be unknown initially since that time may depend on other events that may occur. An example of another event would be the arrival of a rush order that preempts the resource. When the preempt occurs, the entity using the resource relinquishes its control instantaneously. Another example is a failure necessitating repair of the resource. Discrete-event simulations contain activities that cause time to advance. Most discrete-event simulations also contain delays as entities wait. The beginning and ending of an activity or delay is an event.

### 1.2.3 Model Classifications

Several classification categories for models exist. A system we are modeling exhibits **probabilistic** or **stochastic behavior** if an element of chance exists. For example, the path of a hurricane is probabilistic. In contrast, a behavior can be **deterministic**, such as the position of a falling object in a vacuum. Similarly, models can be deterministic or probabilistic. A **probabilistic** or **stochastic model** exhibits random effects, while a **deterministic model** does not. The results of a deterministic model depend on the initial conditions; and in the case of computer implementation with particular input, the output

is the same for each program execution. As we studied this and other modules, we can have a probabilistic model for a deterministic situation, such as a model that uses random numbers to estimate the area under a curve .Figure 6 is depicted the classification of different kinds of models.

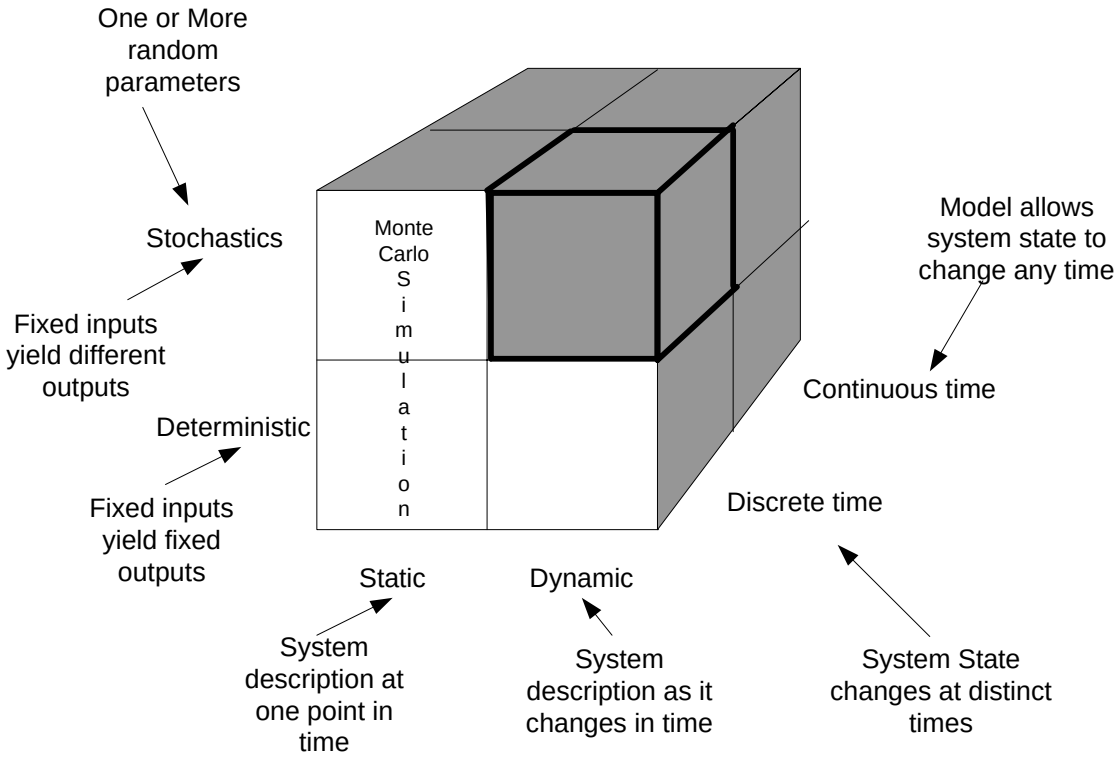



**Figure 6: Classification of Different Types of Model**

### 1.2.3.1 Discrete-Event Simulation Model

Sufficient modeling concepts have been defined so that a discrete event simulation model can be defined as one in which the state variables change only at those discrete points in time at which events occur. Events occur as a consequence of activity times and delays. Entities may compete for system resources, possibly joining queues while waiting for an available resource. Activity and delay times may "hold" entities for durations of time. A discrete-event simulation model is conducted over time ("run") by a mechanism that moves simulated time forward. The system state is updated at each event along with capturing and freeing of resources that may occur at that time.

### 1.2.3.2 Stochastic and Deterministic Systems

**Definitions** A system exhibits **probabilistic** or **stochastic behavior** if an element of chance exists. Otherwise, it exhibits **deterministic behavior**. A **probabilistic** or **stochastic model** exhibits random effects, while a **deterministic model** does not.

**Deterministic:** Randomness does not affect the behaviour of the system. The output of the system is not a random variable.
**Stochastic:** Randomness affects the behaviour of the system. The output of the system is a random variable.

### 1.2.3.3 Static and Dynamic Simulations

We can also classify models as static or dynamic. In a **static model**, we do not consider time, so that the model is comparable to a snapshot or a map. For example, a model of the weight of a salamander as being proportional to the cube of its length has variables for weight and length, but not for time. By contrast, in a **dynamic model**, time changes, so that such a model is comparable to an animated cartoon or a movie. For example, the number of salamanders in an area undergoing development changes with time; and, hence, a model of such a population is dynamic. Many of the models we consider in this text are dynamic and employ a static component as part of the dynamic model.
**Definitions** A **static model** does not consider time, while a **dynamic model** changes with time.

**Static:** A simulation of a system at one specific time, or a simulation in which time is not a relevant parameter for example, Monte Carlo & steady-state simulations.
**Dynamic:** A simulation representing a system evolving over time for examples, the majority of simulation problems.

### 1.2.3.4 Discrete vs. Continuous Systems

When time changes continuously and smoothly, the model is **continuous**. If time changes in incremental steps, the model is **discrete**. A discrete model is analogous to a movie. A sequence of frames moves so quickly that the viewer perceives motion. However, in a live play, the action is continuous. Just as a discrete sequence of movie frames represents the continuous motion of actors, we often develop discrete computer models of continuous situations .
**Definitions** In a **continuous model**, time changes continuously, while in a **discrete model** time changes in incremental steps.

**Continuous: State** variables change continuously as a function of time (Figure 7) and generally analytical method like deductive mathematical reasoning is used to define and solve the system.

**State Variable (S.V.) = f (t)**

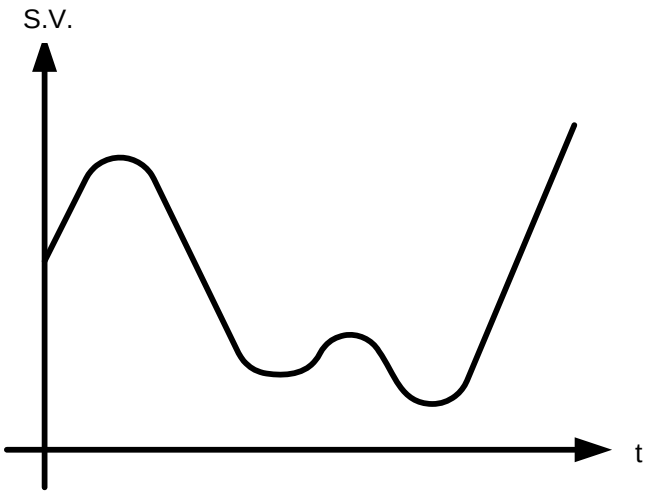



*Figure 7:Continous Model behavior is shown*

**Discrete:** State variables change at discrete points in time (Figure 8) and generally numerical method like computational procedures is used to solve mathematical models.

***State Variable(S.V.) = f(n t)***

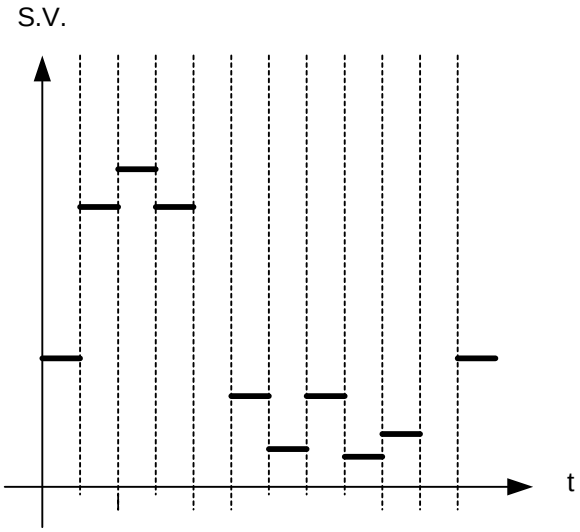



**Figure 8: Discrete Model behavior is shown**

**Examples of Different Systems**

- Queue length at a cash machine: Stochastic, Discrete Time, Discrete System
- The motion of the planets: Deterministic, Continuous Time, Discrete System
- Logic circuit in a computer: Deterministic, Discrete Time, Discrete System

- Flow of air around a car: Deterministic, Continuous Time, Continuous System
- Closing prices of the 30 DAX shares: Stochastic, Discrete Time, Discrete System

#### 1.2.3.5 A Classic Example of Queue at Bank Counter

We see queues at everywhere. Queues are buffers to smooth out differences in arrival rates and service times. Queue Theory is well understood. Closed-form queue-theoretic models can be used to speed up simulations. Deriving results from such models requires simulation. Here are we given a example of queue formed at bank counter (Concept of queue is discussed in more detail in unit IV) .At bank counter customers arrive at random intervals and suppose there is only one cashier .Customers must wait in a queue. Service times at the cashier are also random Measured inter-arrival times (seconds):25, 111, 56, 232, 97, 452, 153, 45,...Measured service times (seconds): 45, 32, 11, 61, 93, 56, 30,...

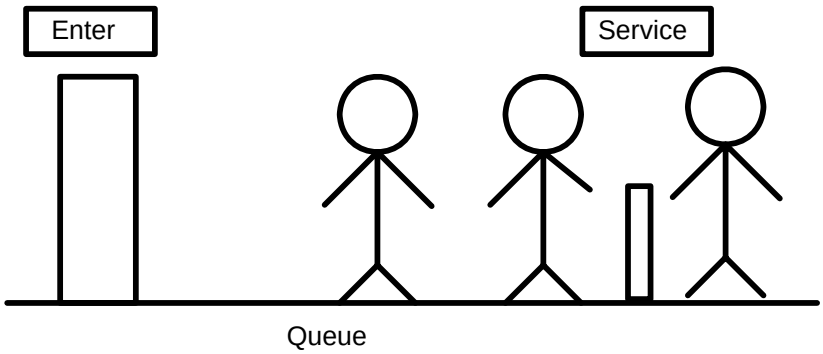



**Figure 7: Classical Example of Queue**

Now compute the average length of the queue and the probability that the cashier is busy.

## 1.3 Computer Workload and Preparation of its Models

Figure 8 outlines the phases of computer simulation development. All simulation studies begin with a specification of objectives and problem formulation. Model development including conceptualization and implementation follows the formulation. Data are collected from the real world early during model development to adequately provide the parameters of the model entities. Decisions need to be made during the implementation phase on the choice of the platform, language, analysis methods, etc. The implemented model must be verified for accuracy and validated for correspondence to the real-world system being represented. The simulation is run several times and statistical analyses of the output data are conducted before a modeler provides recommendations based on the simulation study. The processes involved in simulation modeling and analysis are not strictly linear. An analyst may iterate between different stages in computer simulation development including problem formulation, model abstraction, implementation, verification and validation, and output analysis. The different stages are detailed below

### 1.3.1 Steps of the Modeling Process

The modeling process is cyclic and closely parallels the scientific method and the software development life cycle (SDLC) for the development of a major software project. The process is cyclic because at any step we might return to an earlier stage to make revisions and continue the process from that point.

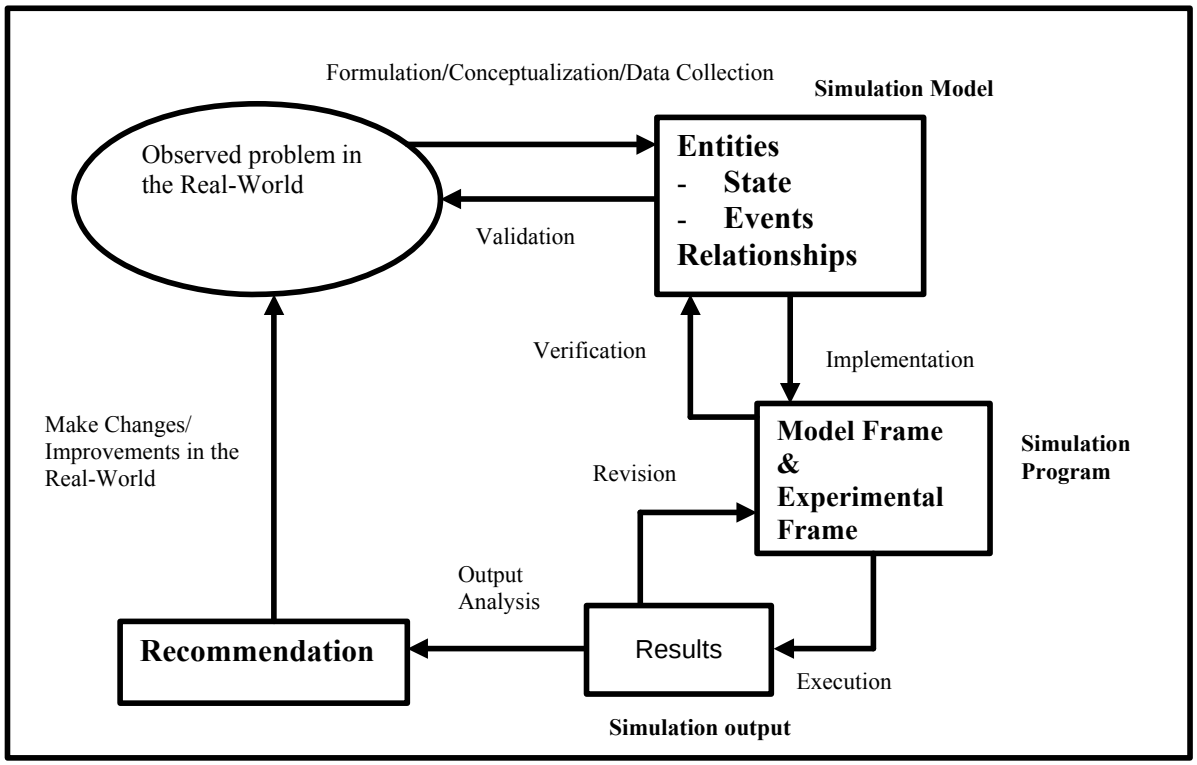



**Figure 8. Phases of Computer Simulation Development and Analysis**

The steps of the modeling process are as follows:

**1. Analyze the Problem**

We must first study the situation sufficiently to identify the problem precisely and understand its fundamental questions clearly. At this stage, we determine the problem's objective and decide on the problem's classification, such as deterministic or stochastic. Only with clear, precise problem identification can we translate the problem into mathematical symbols and develop and solve the model.

**2. Formulate a Model**

A specific set of goals initiate a simulation study. The goals could be for *what-if* analysis of a system being designed or for evaluation of variety of prototypical scenarios of an existing system. For example, in a service industry application, simulations of banking systems can be used to evaluate the tradeoffs associated with having an additional teller. The performance measures of the system being studied must be made explicit during the

problem formulation stage. In the banking system example, performance measures include average customer waiting time, teller utilization, etc. Decisions on whether simulation is an appropriate methodology must also be made carefully during this stage.
In this stage, we design the model, forming an abstraction of the system we are modeling. Some of the tasks of this step are as follows:

**a. Gather Data**

We collect relevant data to gain information about the system's behavior.

**b. Make Simplifying Assumptions and Document them**

In formulating a model, we should attempt to be as simple as reasonably possible. Thus, frequently we decide to simplify some of the factors and to ignore other factors that do not seem as important. Most problems are entirely too complex to consider every detail, and doing so would only make the model impossible to solve or to run in a reasonable amount of time on a computer. Moreover, factors often exist that do not appreciably affect outcomes. Besides simplifying factors, we may decide to return to Step 1 to restrict further the problem under investigation.

**3. Model Abstraction**

Following problem formulation, the analyst abstracts relevant features of the system to be represented in the model. Depending on the goals of the simulation study, the analyst decides on the appropriate level of detail or granularity of the model. Aspects of the system relevant to the simulation must be specified. Once the scope has been sufficiently limited, the system boundaries can be drawn. Bounding the system is a very important analytical step because it defines internal and external factors as well as inputs and outputs. The process of bounding the system can help to reduce the level of complexity by reducing the size of the system in some cases. By drawing a boundary around the system, attention is drawn to the impacts on and by other systems interacting with the system of interest. Several valuable questions can be raised in the process of delineating the problem system. The problem focus is sharpened and some attention is drawn to environmental factors outside the system. Frequently, diagramming techniques such as flow charts are helpful in pictorially representing the system in terms of entities, their behavior, and the interactions between entities. The diagrams can be very useful in communicating about the system and can sometimes reveal additional insights about the problem.
It is very helpful when conceptualizing a system to have a consistent set of questions to answer for each element of the system. These questions provide a parallel structure for the various elements of the system and ensure that all the relevant information is obtained. Forming these general questions stimulates a broader view of the elements of the system and stimulates comparisons and contrasts among them. In answering these general questions about the system, some simplifying assumptions can be made initially. There are two basic types of assumptions about systems: structural assumptions and data assumptions. Structural assumptions are made regarding the internal operation of the system. They concern differences between the way a system is designed to work and the

way it actually operates in practice under a variety of conditions. Some steps may be shortened or bypassed completely under some circumstances for the sake of expediency. Alternate routing can occur in cases of blockage or excessive queuing. More realistic factors can be added to the model gradually as the development proceeds. Data assumptions are made with respect to the entities being processed. Inputs may cycle through peak and slack periods. Individual times may vary for different entities in the same process. These can be included later when they have been more accurately determined. Assumptions made in the problem formulation should be explicitly listed so they can be addressed eventually in a meaningful manner.

**4. Determine Variables and Units**

We must determine and name the variables. An **independent variable** is the variable on which others depend. In many applications, time is an independent variable. The model will try to explain the **dependent variables**. For example, in simulating the trajectory of a ball, time is an independent variable; and the height and the horizontal distance from the initial position are dependent variables whose values depend on the time. To simplify the model, we may decide to neglect some variables (such as air resistance), treat certain variables as constants, or aggregate several variables into one. While deciding on the variables, we must also establish their units, such as days as the unit for time.

**a. Establish Relationships Among Variables and Submodels**

If possible, we should draw a diagram of the model, breaking it into submodels and indicating relationships among variables. To simplify the model, we may assume that some of the relationships are simpler than they really are. For example, we might assume that two variables are related in a linear manner instead of in a more complex way.

**b. Determine Equations and Functions**

While establishing relationships between variables, we determine equations and functions for these variables. For example, we might decide that two variables are proportional to each other, or we might establish that a known scientific formula or equation applies to the model. Many computational science models involve differential equations, or equations involving a derivative.

**5. Solve the Model**

This stage implements the model. It is important not to jump to this step before thoroughly understanding the problem and designing the model. Otherwise, we might waste much time, which can be most frustrating. Some of the techniques and tools that the solution might employ are algebra, calculus, graphs, computer programs, and computer packages. Our solution might produce an exact answer or might simulate the situation. If the model is too
complex to solve, we must return to Step 2 to make additional simplifying assumptions or to Step 1 to reformulate the problem.

**6. Model Implementation**

The conceptual model generated during the earlier phase must be implemented in the form of a simulation program. During implementation, an analyst can use a simulation language such as SIMAN or GPSS, or standard programming languages such as C, C++, or Java, or special simulators tailored for specific applications. Typically, most simulation languages provide software constructs to represent entities, to perform queuing utilities, and standard statistical analysis. The SIMAN simulation language also orients the analyst to make decomposition between the model frame, which is used to represent the structure and overall behavior of the system and experimental frame, which is used to store data used by the model. Some languages have a graphical front end to configure a system simulation. The graphical environment is intended for analysts without programming background to rapidly simulate and analyze a specific system. It takes longer to use standard programming languages to apply for simulation, but they provide greater levels of control in representing complex decision making behavior. The specific simulation software or language selected is dependent on the nature of the application, expertise of the analysts, and availability of appropriate hardware and software. The fundamental problem in model implementation is to translate the conceptual model to a simulation program using the constructs of a simulation or software package. Model verification and validation typically follow the model implementation phase.

### 7. Verify and Interpret the Model's Solution

Once we have a solution, and the model's solution is used, it may be necessary or desirable to make corrections, improvements, or enhancements. In this case, the modeler again cycles through the modeling process to develop a revised solution. We should carefully examine the results to make sure that they make sense (verification) and that the solution solves the original problem (validation) and is usable. The process of **verification** determines if the solution works correctly, while the process of **validation** establishes if the system satisfies the problem's requirements. Thus, verification concerns "solving the problem right," and validation concerns "solving the right problem." Testing the solution to see if predictions agree with real data is important for verification. We must be careful to apply our model only in the appropriate ranges for the independent data. For example, our model might be accurate for time periods of a few days but grossly inaccurate when applied to time periods of several years. We should analyze the model's solution to determine its implications. If the model solution shows weaknesses, we should return to Step 1 or 2 to determine if it is feasible to refine the model. If so, we cycle back through the process. Hence, the cyclic modeling process is a trade-off between **simplification** and **refinement**. For refinement, we may need to extend the scope of the problem in Step 1. In Step 2, while refining, we often need to reconsider our simplifying assumptions, include more variables, assume more complex relationships among the variables and submodels, and use more sophisticated techniques.

Although we described the modeling process as a sequence or series of steps, we may be developing two or more steps simultaneously. For example, it is advisable to be compiling the report from the beginning. Otherwise, we can forget to mention significant

points, such as reasons for making certain simplifying assumptions or for needing particular refinements. Moreover, within modeling teams, individuals or groups frequently work on different submodels simultaneously. Having completed a submodule, a team member might be verifying the submodule while others are still working on solving theirs.
The modeling process is a creative, scientific endeavor. As such, a problem we are modeling usually does not have one correct answer. The problems are complex, and many models provide good, although different, solutions. Thus, modeling is a challenging, open-ended, and exciting venture.
Validation is the process of assuring that the conceptual model accurately represents the behavior of the real system. Verification is the process of assuring that the implemented model accurately represents the conceptual model. These two processes are theoretically distinct but are closely related in practice. The initial conceptual model should have high *face validity*. Input should be sought from as wide a range as possible of people knowledgeable about the system. There are at least three reasons for this. Primarily, people who work with the system in different ways have different knowledge about the system. Some of this knowledge overlaps with others, but some are unique to a particular perspective. The unique perspectives complete the system concept and correct misconceptions. Secondly, the overlapping areas of knowledge provide crosschecks of the various inputs for consistency. Thirdly, participation in the modeling process reinforces confidence in the simulation. It provides the users the opportunity to question and critique the conceptual model. Involvement enhances acceptance and understanding among the users of the real system being modeled. Without end-user involvement, user skepticism and resistance can thwart improvements.

After the face validity of the conceptual model has been established the assumptions made about the model must be examined. Estimates by those experienced with the system are the best initial values for both types of assumptions. However, careful data collection and statistical analyses are required for refinements. The first step in the input data analysis is the identification of the appropriate statistical distribution for the data. Second, the parameters relevant to that distribution must be estimated from the sampled data. Finally, the fit of the selected distribution to the data can be verified by applying appropriate statistical tests such as the Kolmorgorov – Smirnov (K-S) test or chi-squared test.

Ultimately, the model must accurately predict the system performance for a range of input conditions. The simulation should be robust enough to yield accurate results when assumptions and parameters are varied. The most conclusive test for the model is the simulation of the system under conditions where the outputs of the real system are known. The focus during this process is on the overall transformation of the inputs into outputs. The outputs of the simulation can then be compared with the historical data. This testing can be only made for situations where historical data are available.

**8. Execution**

Most simulations represent random or stochastic characteristics where different quantitative values are obtained for the same model with varying random values. For such simulations, called non-deterministic systems, the simulation model must be executed several times and statistical analysis must be performed on the simulation output to assess variability of the simulation. From an output analysis perspective, there are two types of execution modes: terminating and steady state. The terminating mode is appropriate for systems that start and run for a time then stops for some period. An example of this kind of operation is a store that closes overnight. A steady state simulation is suitable for a system that operates continuously, such a power generating plant or a hospital emergency room. The widest possible set of initial conditions should be tried in both cases.

Initial conditions can have a profound effect on the results of simulations. Initial conditions include such things as whether or not the system resources start up in the busy or idle state. Not only should all possible sets of initial conditions be tried for a given set of conditions, but also the simulations should be run several times for each set of conditions. For terminating simulations, this means that several runs are required. In the case of a steady state simulation, data samples should be taken from several subintervals in each run. The subintervals should be of equal length in all runs for valid statistical comparisons to be made.

**9. Output Analysis**

A simulation is a statistical experiment that imitates the real system. Appropriate statistical methods should be applied to determine the degree of correspondence between the outputs of the model and those of the real system. These methods will vary depending upon whether a terminating or steady state simulation is being analyzed. The basic goal here is to determine the variability in the estimators chosen to evaluate the system. In general steady state simulations are more difficult to analyze due to the effects of initial conditions and choice of run time. For steady-state systems, data should not be used if it is collected before statistical equilibrium is reached. Initial condition bias can lead to erroneous conclusions, even for long run times and multiple runs. The estimators may have smaller variation, but they may converge to the wrong value. Typically, a type of hypothesis testing is conducted using confidence intervals as a function of variability of the output data from multiple simulation runs.

**10. Report on the Model**

Reporting on a model is important for its utility. Perhaps the scientific report will be written for colleagues at a laboratory or will be presented at a scientific conference. A report contains the following components, which parallel the steps of the modeling process:

**a. Analysis of the problem**

Usually, assuming that the audience is intelligent but not aware of the situation, we need to describe the circumstances in which the problem arises. Then, we must clearly explain the problem and the objectives of the study.

**b. Model design**

The amount of detail with which we explain the model depends on the situation. In a comprehensive technical report, we can incorporate much more detail than in a conference talk. For example, in the former case, we often include the source code for our programs. In either case, we should state the simplifying assumptions and the rationale for employing them. Usually, we will present some of the data in tables or graphs. Such Figures should contain titles, sources, and labels for columns and axes. Clearly labeled diagrams of the relationships among variables and submodels are usually very helpful in understanding the model.

**c. Model solution**

In this section, we describe the techniques for solving the problem and the solution. We should give as much detail as necessary for the audience to understand the material without becoming mired in technical minutia. For a written report, appendices may contain more detail, such as source code of programs and additional information about the solutions of equations.

**d. Results and conclusions**

Our report should include results, interpretations, implications, recommendations, and conclusions of the model's solution. We may also include suggestions for future work.

**11. Recommendation**

Ultimately the results are used to decide what changes to make to the real system, if any. Naturally, there is a cost associated with changing a system. Any changes will be judged in the long run by the savings or additional revenue generated. Other systems that interact with the system being simulated may be affected by changes. After the simulation has been done and the data has been thoroughly studied, the analyst must look beyond the system on which the attention has been focused. In some cases a less costly change in some other system may yield more cost-effective improvements. This possibility should be detected in the problem analysis phase preceding simulation, but sometimes elements in the systems environment are overlooked. Finally, although planning and implementing the changes may not be the responsibility of the analysts, any recommendations for change should be described in as much detail as possible. Cost and impact analysis along with a detailed description forms a solid proposal for change. Any changes made to the real world must be monitored and data must be collected from the

real system to feed into the simulation model. The usefulness of the model does not end when the analysis is concluded. The model represents an investment of resources and can continue to provide useful data when changes to the system are proposed again.

## 1.4 Summary

Simulation is very powerful, problem solving technique. Its applicability is so general that it would be hard to point out disciplines or system to which simulation has not been applied. The basics idea behind simulation is simple, namely model the given system by means of some equations and then determine its time dependent behaviour. In Simulation we makes a model of conceptual model and then results are compared with real system. Normally simulation is used when either an exact analytic expression for the behaviour of the system under investigation is not available or the analytic solution is too time consuming. Simulation is considered as interdisciplinary subject because it uses concepts from mathematics computer science and application field. A model is a representation of an actual system. Models can be of different kinds. Discrete-Event Simulation Model is defined as one in which the state variables change only at discrete points in time at which events occur. A deterministic system is defined as in which randomness does not affect the behaviour of the system. A stochastic system is defined as in which randomness affects the behaviour of the system. In order to process a simulation of an event first we have to perform formulation of the problem, and then we have to implement the defined model in any suitable programming language. Finally we have to perform verification and validation and then analysis of output results.

## 1.5 Key Words

System, Model, Base Model, Simulation, Continuous, Discrete, State Variables, Dynamic, Digital Computer, Statistical Sampling, Stochastic Systems, Numerical Techniques, Monte Carlo Method, Graphical User Interfaces, Object- Oriented Programming, Optimization, Entities, Attributes, FIFO, LIFO, Statistical Analysis Software (SAS), Software Development Life Cycle (SDLC), Probability, Differential Equation, Face Validity.

## 1.6 Self Assessment Questions

Q1. Give an example of a nonstochastic simulation?
Q.2 Give an example of each

a. Stochastic Model
b. Continuous Model
c. Discrete Model
d. Static Model
e. Dynamic Model

Q.3 What is Model? Define some model of surrounding events.
Q.4 What is a system? Define kinds of system.
Q.5 What is the difference between static and dynamic model?
Q.6. What is the difference between continuous and discrete model?
Q.7 What is simulation? Give some advantage and disadvantage of simulation.
Q.8 Write some application of simulation.
Q.9 How to Build and Apply Computer Simulations? Explain.
Q.10 What is meant by the "System State" in a simulation.
Q.11 Compare and contrast the modeling process with the scientific method: Make observations; formulate a hypothesis; develop a testing method for the hypothesis; collect data for the test; using the data, test the hypothesis; accept or reject the hypothesis.
Q.12 Compare and contrast the modeling process with the software life cycle: Analysis, design, implementation, testing, documentation, and maintenance.
Q.13 What do you understand by the term face Validity of a conceptual model?

## 1.7 Reference /Suggested Reading


1. *Proceedings of the 1999 Winter Simulation Conference,* Jerry Banks, **Introduction to Simulation**
2. Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, second edition.
3. Banks, Carson, Nelson & Nichol, *Discrete Event System Simulation*, Prentice Hall.
4. G.Gorden, "System Simulation",PHI.
5. N. Deo , " System Simulation", PHI.
6. Giordano, Frank R., Maurice D. Weir, and William P. Fox. 2003. *A First Course in Mathematical Modeling*. 3rd ed. Pacific Grove, Calif.: Brooks/Cole-Thompson Learning.

**Subject : System Simulation and Modeling** **Author : Jagat Kumar**
**Paper Code: MCA 504** **Vetter : Dr. Pradeep Bhatia**
**Lesson : Verification and Validation of Models**
**Lesson No. : 02**

**Structure**


## 2.0 Objective

The area of experimentation and results analysis for simulation models is briefly introduced here. By the end of this module you will learn the verification and validation techniques to compare the defined model with real system's data.

## 2.1 Introduction

How do we know that the model we have used, is an accurate representation of the system being simulated? This is an important question and must be answered satisfactorily before a simulation study can be made use of. The area of experimentation and results analysis for simulation models is well developed and a range of rules and guiding methods can be found in the literature, e.g. are available. Many of the techniques developed are here to ensure that dangerous mistakes are not made when analyzing and interpreting the results. In fact , without establishing the validity of the model , if we accept the(erroneous) simulation results the consequences may be disastrous. Put simply the power of modern simulation software to generate large quantities of data can leave the user with the false sense of security that the results generated are credible and truly representative of the system under study. Like all modelling techniques care needs to be exercised.
What is a valid model? Since no simulation model will duplicate the given system in every detail it is not an appropriate question to ask if a simulator is a 'true' model of a real system. We should only ask if the model is a 'reasonable' approximation of the real

system. The acceptable levels of reasonableness and approximation will vary from system to system and simulation to simulation. There is no universally acceptable criteria for accepting a simulation model as a valid representation. There are only guidelines that aid in establishing confidence in the model.

There are a number of phases as shown in Figure 1, for checking a simulation model prior to experimental analysis:

- **Verification** - the accuracy of transforming a problem formulation into a model (specification) deals with building the **model right**
- **Validation** - model behaves with satisfactory accuracy consistent with the study objectives deals with building the **right model**

The above are essential checks performed prior to analysis of the model and are used to establish what is known as model credibility. Verification and validation is shown in Figure 2, as cause - effect and input – output terms.

A simulation run typically starts in the empty and idle state. The run is therefore characterised by a "run-in" phase followed by a "steady state" phase, see Figure 1. The run-in phase is generally ignored and is only used for investigating the effects of transient conditions such as starting up a new factory or performing radical changes within an existing facility.

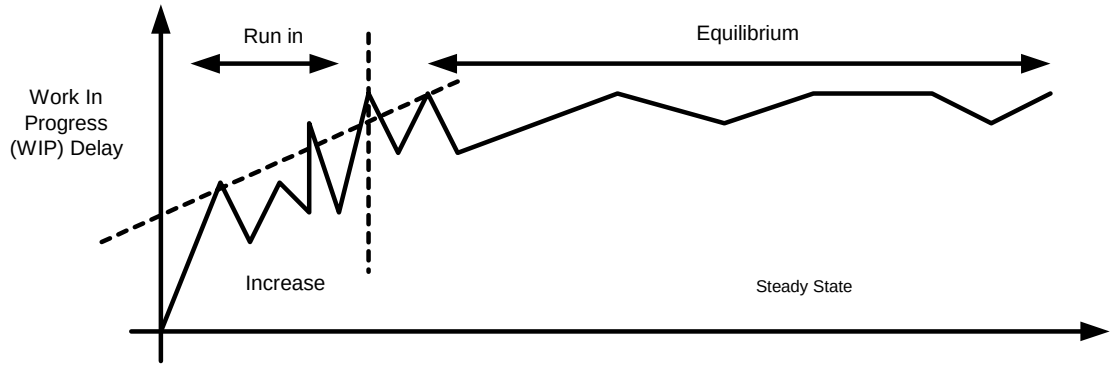



**Figure 1. The two key phases of a simulation run**

Typically the steady state phase is of greater interest. At this stage checks must be made to ensure no long term trends exist, such as continual build up of stock in the factory, that suggest the model (hence the real system) will be unstable and unworkable.
Generally what is known as multiple replications is performed. This is where the model is run several times. Each time the random number generators are set to provide different sequences of random numbers, e.g. in case of manufacturing process the breakdown patterns of machines are different and the points at which material is scrapped is different. This allows confidence that the results being compiled represent the average and the range of conditions that are likely and therefore play down ‘freak’ or ‘unusual’ behaviour.

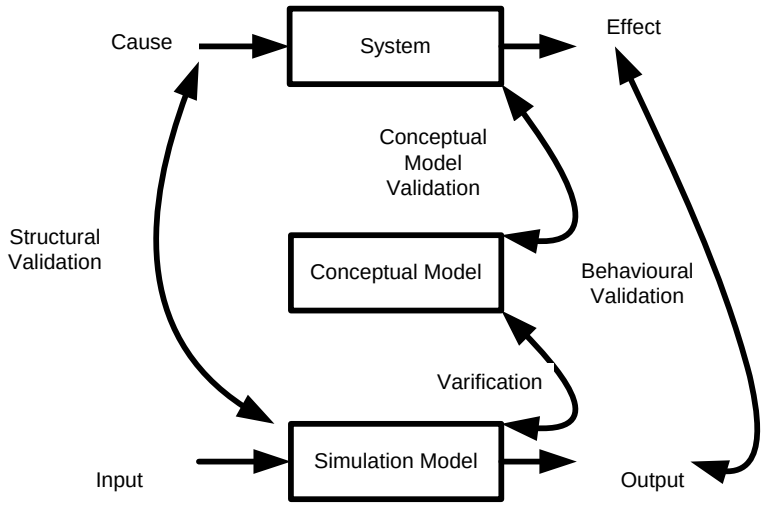



**Figure 2 Verification and Validation Activities**

## 2.2 Verification and Validation

The validation efforts can be grouped into two parts

1. validation of the abstract model itself
2. validation of its implementation

The first part consists of examining all assumptions, which transform the real world system into the conceptual model. A great deal of judgment and an intimate knowledge of the real system are involved in this step. The validation of the abstract model is often highly subjective. Testing the validity of an implementation is a more objective and easier task. It consists of checking the logic, the flowchart, and the computer program to ensure that the model has been correctly implemented.

The presentation of an experimental frame, which is shown in Figure 2, enables a rigorous definition of model *validity*. Let us first postulate the existence of a unique *Base Model*. This model is assumed to accurately represent the behavior of the Real System under *all* possible experimental conditions. This model is *universally valid* as the data $\mathbf{D_{RealSystem}}$ obtainable from the Real System is always equal **(symbol ≡ is used for equality)** to the data $\mathbf{D_{BaseModel}}$ obtainable from the model.

$$\mathbf{D_{BaseModel}} \equiv \mathbf{D_{RealSystem}} \quad \text{----------(i)}$$

A Base Model is distinguished from a **Lumped Model** by the limited experimental context within which the latter accurately represents Real System behavior.
A particular **experimental frame *E*** may be applicable to a real system or to a model. In the first case, the data potentially obtainable within the context of *E* are denoted by

$D_{RealSystem} \| E$. In the second case, obtainable data are denote by **Dmodel ‖ E**. With this notation, a model is valid for a real system within Experimental Frame *E* if

$$D_{LumpedModel} \| E \equiv D_{RealSystem} \| E \qquad \text{----------(ii)}$$

The data equality ≡ *m*ust be interpreted as equal to a certain degree of accuracy. It shows how the concept of validity is not absolute, but is related to the experimental *context* within which Model and Real System *behavior* are compared and to the *accuracy metric* used.
One typically distinguishes between the following types of model validity.

**Replicative Validity** concerns the ability of the Lumped Model to *replicate* the input/output data of the Real System. With the definition of a Base Model, a **Lumped Model** is replicatively valid in Experimental Frame *E* for a Real System if

$$\mathbf{D_{LumpedModel} \| E \equiv D_{BaseModel} \| E} \qquad \text{----------(iii)}$$

**Predictive Validity** concerns the ability to identify the *state* a model should be set into to allow *prediction* of the response of the Real System to *any* (not only the ones used to identify the model) input segment. A Lumped Model is predicatively valid in Experimental Frame *E* for a Real System if it is replicatively valid and

$$F_{LumpedModel} \| E \subseteq \mathbf{F}_{BaseModel} \| E \qquad \text{----------(iv)}$$

where $F_S$ is the set of I/O functions of system *S* within Experimental Frame *E*. An I/O function identifies a *functional relationship* between Input and Output, as opposed to a general non-functional *relation* in the case of replicative validity.

**Structural Validity** concerns the *structural relationship* between the Real System and the Lumped Model. A Lumped Model is structurally valid in Experimental Frame *E* for a Real System if it is predictively valid and there exists a **morphism** Δ from Base Model to Lumped Model within frame *E*.

$$\mathbf{LumpedModel \| E \Delta BaseModel \| E} \qquad \text{----------(v)}$$

When trying to assess model validity, one must bear in mind that one only observes, at any time *t*, $\mathbf{D_{RealSystem}(t)}$ , a subset of the potentially observable data $D_{RealSystem}$. This obviously does not simplify the model validation enterprise.
Whereas assessing model validity is intrinsically impossible, the *verification* of a *model implementation* can be done rigorously. A *simulator* implements a lumped model and is thus a source of obtainable data $\boldsymbol{D}_{\mathbf{Simulator}}$ . If it is possible to prove (often by design) a

structural realtionship (morphism) between Lumped model and Simulator, the following will hold unconditionally

$$D_{Simulator} \equiv D_{LumpedModel} \qquad \text{----------(vi)}$$

Before we go deeper into predictive validity, the relationship between different *refinements* of both Experimental Frames and models is elaborated. In Figure 3, the *derived from* relationship for Experimental Frames and the *homomorphism.*

Relationship for Models is depicted. If we think of an Experimental Frame as a formal representation of the context within which the model is a valid representation of the dynamics of the system, a more restricted Experimental Frame means a more specific behaviour. It is obvious that such a restricted Experimental Frame will match far more models than a more general Experimental Frame. Few models are elaborate enough to be valid in a very general input/parameter/performance range. Hence, the large number of applies to (*i.e.,* match) lines emanating from a restricted Experimental Frame. The homomorphism between models means that, when modifying/transforming a model (*e.g.,* adding some non-linear term to a model), the simulation results (*i.e.,* the behaviour) within the same experimental frame must remain the same.

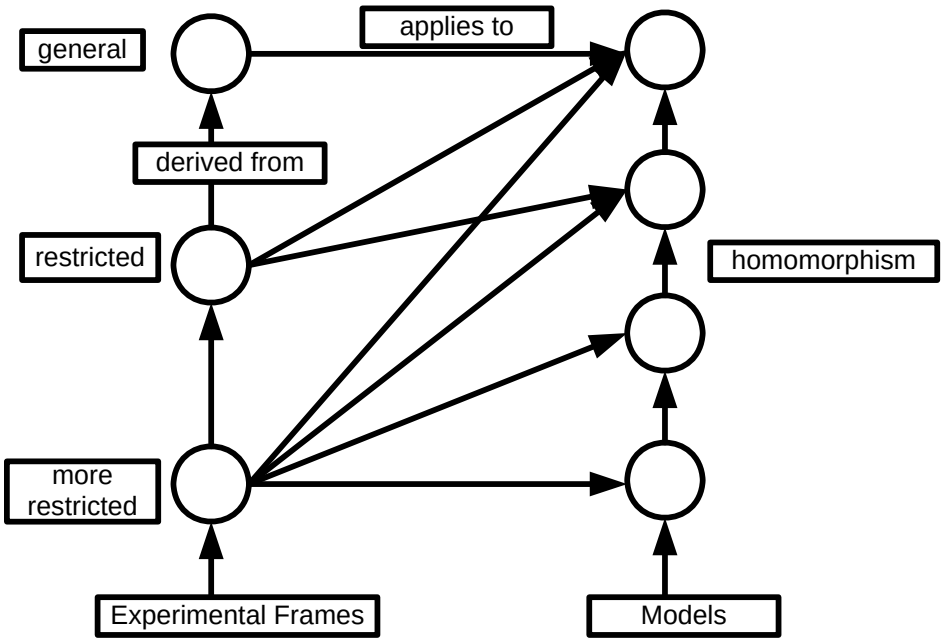



**Figure 3 : Experimental Frame – Model Relationship**

Though it is meaningful to keep the above in mind during model development and use, the highly non-linear nature of many continuous models makes it very difficult to automate the management of information depicted in Figure 3. Non-linear behaviour makes it almost impossible, based on a model or experimental frame symbolic representation, to make a statement about the area in state-space, which will be covered (*i.e.,* behaviour). A pragmatic approach is to

1. Let an expert indicate what the different relations are. This is based on some insight into the nonlinear dynamics. Such expert knowledge can be built from a large number of conducted experiments.
2. Constantly with each experiment validate the expert information.

A crucial question is whether a model has predictive validity, is it capable not only of reproducing data which was used to choose the model and parameters but also of predicting new behavior? The predictive validity of a model is usually substantiated by comparing new experimental data sets to those produced by simulation, an activity known as model validation. Due to its special importance in the communication between model builders and users, model validation has received considerable attention in the past few decades. The comparison of the experimental and simulation data are accomplished either subjectively, such as through graphical comparison, Turing test, or statistically, such as through analysis of the mean and variance of the residual signal employing the standard $F$ statistics, multivariate analysis of variance regression analysis, spectral analysis, autoregressive analysis, autocorrelation function testing, error analysis, and some non-parametric methods.

The above-mentioned methods are designed to determine, through comparison of measured and simulated data, the validity of a model. As one might intuitively expect, different modelling errors usually cause the behavior of the model to deviate in different ways from that of the real system or, in other words, different modelling errors correspond to different pattern in the error signal, the difference between experimental data and simulated data. These patterns if extractable, can obviously be used to identify the modelling errors.

## 2.3 Comparing Model Data with Real System Data

After development of suitable model of defined problem and simulation of defined model, we have to perform the comparison of output data with real system data.

### 2.3.1 Validating Existing Systems

When the simulated system exists in real life, then the most obvious and the best approach is to use the real world inputs to the model and compare its outputs with that of the real world inputs to the model and compare its output with that of the real system. This process of validation is straightforward enough in principal but may present some difficulties when carried out. Firstly, it may not always be easy to obtain input and output data from a real life system without distributing it. Secondly, even if we could get actual input output of an existing model it would not generally be for very long periods. Since the data are usually probabilistic. For small lengths of simulation runs the variability of the model output would be large. Therefore designing test that work with small samples is difficult. What usually is done is to simulate the model several times (replicate) with different sequences of random numbers and obtain the range of variation amongst these. Then, if the model is valid, the real output should lie somewhere in the middle of the range of model output. The third problem is to establish that the model output and the real system outputs are ‘practically’ from the same population.

If the outputs to be compared are sample means (e.g., average queue length, waiting time, idle time), one could use any of a number of statistical tests **(called 'goodness' of fit tests)** available to measure the discrepancy between the two outputs (i.e., model output and the real system output). One such test is **chi-square test**. Others are **Kolmogorov-Smirnov test, Cramer - von Mises test** and the **Moments test**. One could also use hypothesis testing to determine if there is any significant difference between, say the average of the independent set of observations.

### *2.3.2 Validating Fist Time Model*

If a model is intended to describe a proposed or hypothethetical system (which does not exist at present or did not exist in the past) then the task of validation is even more difficult. There are no historical data available to compare its performance with. Since hypothetical system are , by their very nature , based upon assumptions it is the validity of these assumptions the simulation model is dependent on. A number of guidelines for testing validity of such system have been found useful. These are as

1. **Subsystem Validity:** A model itself may not have any existing system to compare it with, but it may consist of known subsystem each of that validity can be tested separately.
2. **Internal validity:** One tends to rejects a model if it has a high degree of internal variability. A stochastic system with high variance due to its internal processes will obscure changes in the output due to input changes. The test can be performed by replicating a simulation run with several different random number sequences and then computing the variance of the outputs. If the variance is too high we reject the model.
3. **Sensitivity Analysis:** Sensitivity analysis consists of systematically varying the values of parameters or the input variables one at a time (while keeping all others constant) over some range of interest and observing the effect upon the model's response. Sensitivity analysis will tell to which parameters the system is more sensitive. The parameters to which the system response is relatively insensitive, we need not pay very close attention to. The knowledge how far the assumed parameters values could be from the true one without significantly affecting the response helps building our confidence in the model.
4. **Face Validity:** If the model goes against the common sense and logic, it should be rejected (even if it behaves like the real system). If those with experience and insight into similar system do not judge the model as reasonable, it has to be rejected.

   These and other validation tests do not completely validate a model. While failure to pass a validation test would results in rejection of a model, passing these tests does not guarantee that the model is valid. It only builds up our confidence in the model.

   Ideally errors of a modeling should be separated from the errors in its implementation (programming errors, etc.) by first validating the abstract

mathematical model before writing a simulator for it. In practice, however, it is really possible to check the validity of the mathematical model without examining its computer version. This is because of the mathematical intractability of a model, which was the reason for simulating it in the first place.

Although validation is often messy, expensive and time consuming, involves subjective ness and judgments and is rarely conclusive, it must always be attempted. However inconclusive, it does provide a check against grosser errors and gives us confidence to use the simulation results for decision making.

## 2.4 Summary

Verification and validation of defined system is very critical and it provides the testability to defined system against the real system. Verification is defined as the accuracy of transforming a problem formulation into a model (specification) deals with building the model right while Validation is defined as the defined model behaves with satisfactory accuracy consistent with the study objectives deals with building the right model. Without establishing the validity of the model, simulation results can be erroneous and their consequences may be disastrous.

## 2.5 Keywords

Verification, Validation, System, Model, Conceptual Model, Base Model, Lumped Model, Real System , Kolmogorov- Smirnov test (KS Test), Cramer -von Mises test, Moments test , Goodness of Fit, Face Validity, Replicate Validity, Predictive Validity, Structural Validity, Work-in-Progress (WIP).

## 2.6 Self Assessment Questions

Q.1 Two similar terms used in the steps of a simulation study are “verification” and “validation.” One of them refers to the debugging of the simulation code itself. Which one refers to the process of insuring that the model is a correct representation of the system?
Q.2 What do you understand the term “Model Validation and Verification”? Explain.
Q.3 What do you understand by the term Face Validity of a Conceptual Model?
Q.4 What is the difference between Validation and Verification?
Q.5 Why Validation is so important in Modelling and Simulation?
Q.6 Give some advantages and disadvantage of Validation in Simulation.

## 2.7 Reference /Suggested Reading


7. *Proceedings of the 1999 Winter Simulation Conference,* Jerry Banks, Introduction to Simulation.
8. Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, second edition, 2000.

*9.* Banks, Carson, Nelson & Nichol, *Discrete Event System Simulation*, Prentice Hall.
*10.* N. Deo , " System Simulation", Prentice Hall of India.

**Subject : System Simulation and Modeling** **Author : Jagat Kumar**
**Paper Code: MCA 504** **Vetter : Dr. Pradeep Bhatia**
**Lesson** : **Differential Equations In Simulation**
**Lesson No. : 03**

**Structure**

## 3.0 Objective

The main objective of this unit to introduce the concepts of ordinary and partial differential equations in continuous system modeling. By the end of this module we will learn about different kinds of differential equations and their application in modeling. We will also learn the uses of simulation in education and training.

## 3.1 Introduction

Modeling of a system whether system is continuous or discrete heavily used the concepts of Ordinary Differential Equation (ODE), Partial Differential Equation (PDE) and probability and statistics. Even we can say that without these two branch of mathematics modeling is just impossible.
Continuous processes occur everywhere as we will learn in unit V. Here, we are interested in cases with discrete variables, some examples of continuous process are

- An object falling to the ground
- The motion of the planets orbiting the sun
- The current and voltage in an electrical circuit
- The level of alcohol in my blood on January 1st, 2005
- The populations of a predator and its prey

In almost all above cases, the relationships between the variables and its rate of change i.e. its derivative are defined by an Ordinary Differential Equations (ODEs) are very important in all branches of Science and Engineering. ODEs form the basis for the simulation of almost all continuous phenomena. Understanding ODEs is essential for understanding natural and technical processes.

## 3.2 Ordinary Differential Equations

A **differential equation** is an equation involving an unknown function and its derivatives. The **order** of the differential equation is the order of the highest derivative of the unknown function involved in the equation.

$$\frac{dy}{dx} = f(y,t) \qquad \text{----------(i)}$$

In addition, we usually know the value of y (0) is $y(0) = y_0$.

A **linear differential equation** of order *n* is a differential equation written in the following form

$$a_n(x)\frac{d^n y}{dx^n} + a_{n-1}(x)\frac{d^{n-1} y}{dx^{n-1}} + \ldots\ldots\ldots + a_1(x)\frac{d\ y}{dx} + a_0(x)y = f(x) \text{----------(ii)}$$

where $a_n(x)$ is not the zero function. Note that some may use the notation $y', y'', y''', y^{(4)}, ....$ for the derivatives. A linear equation obliges the unknown function $y$ to have some restrictions. Indeed, the only operations which are accepted for the variable $y$ are:

**(i)** Differentiating $y$
**(ii)** Multiplying $y$ and its derivatives by a function of the variable $x$
**(iii)** Adding what you obtained in (ii) and let it is equal to a function of $x$.

There are several issues related to differential equation solution whether a differential equation have a solution? Does a differential equation have more than one solution? If yes, how can we find a solution, which satisfies particular conditions? A problem in which we are looking for the unknown function of a differential equation where the values of the unknown function and its derivatives at some point are known is called an **initial value problem** (in short IVP). If no initial conditions are given, we call the description of all solutions to the differential equation the **general solution**.

## *3.2.1 Modeling via Differential Equations*

One of the most difficult problems that a scientist deals with in his everyday research is "How do I translate a physical phenomenon into a set of equations which describes it?"

It is usually impossible to describe a phenomenon **totally**, so one usually strives for a set of equations which describes the physical system **approximately** and **adequately**.

In general, once we have built a set of equations, we compare the data generated by the equations with real data collected from the system (by measurement). If the two sets of data "agree" (or are close), then we gain confidence that the set of equations will lead to a good description of the real-world system. For example, we may use the equations to make predictions about the long-term behavior of the system. It is also important to keep in mind that the set of equations stays only "valid" as long as the two sets of data are close. If a prediction from the equations leads to some conclusions which are by no means close to the real-world future behavior, then we should modify and "correct" the underlying equations. As you can see, the problem of generating "good" equations is not an easy exercise.

Note that the set of equations is called a **Model** for the system.

**How do we build a Model?**

The basic steps in building a model are as

**Step 1:** Clearly state the assumptions on which the model will be based. These assumptions should describe the relationships among the quantities to be studied.
**Step 2:** Completely describe the parameters and variables to be used in the model.

**Step 3:** Use the assumptions (from Step 1) to derive mathematical equations relating the parameters and variables (from Step 2).

The best example of mathematical modeling is the one related to population growth problems given below (Example 2). Keep in mind that this problem has many ramifications ranging from population explosion to extinction phenomena.

If there is no analytic solution for systems of ODEs, we are forced to integrate using numerical methods. The numerical integration of ODEs forms the most important part of continuous simulation.

**Example 1: Balance Equations**

Most ODEs are balance equations. A balance equation basically says

Change = Increase – Decrease

For example an ODE for the amount of water x in a tank

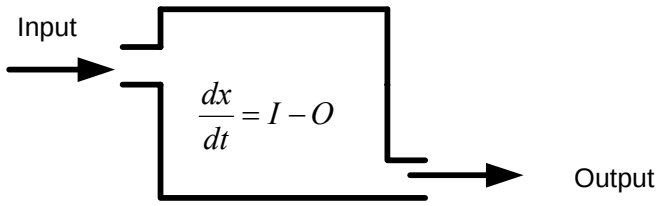



Figure 1: Water tank Problem Model by Balance Differential Equation

## *Example 2: Population Dynamics*

Here are some natural questions related to population problems:

- What will the population of a certain country be in ten years?
- How are we protecting the resources from extinction?

More can be said about the problem but, in this brief discussion we will not discuss them in detail. In order to illustrate the use of differential equations with regard to this problem we consider the easiest mathematical model offered to govern the population dynamics of a certain species. It is commonly called **the exponential model**, that is, the rate of change of the population is proportional to the existing population. In other words, if $P(t)$ measures the population, we have

$$\frac{dP}{dt} = kP$$

where the rate $k$ is constant. It is fairly easy to see that if $k > 0$, we have growth, and if $k < 0$, we have decay. This is a linear equation which solves into

$$P(t) = P_0 e^{kt}$$

where $P_0$ is the initial population, i.e. $P(0) = P_0$. Therefore, we conclude the following:

- if $k>0$, then the population grows and continues to expand to infinity, that is,

$$\lim_{t \to +\infty} P(t) = +\infty$$

- if $k<0$, then the population will shrink and tend to 0. In other words we are facing extinction.

Clearly, the first case, $k>0$, is not adequate and the model can be dropped. The main argument for this has to do with environmental limitations. The complication is that population growth is eventually limited by some factor, usually one from among many essential resources. When a population is far from its limits of growth it can grow exponentially. However, when nearing its limits the population size can fluctuate, even chaotically. Another model was proposed to remedy this flaw in the exponential model. It is called the logistic model (also called Verhulst-Pearl model). The differential equation for this model is

$$\frac{dP}{dt} = kP\left(1 - \frac{P}{M}\right)$$

where $M$ is a limiting size for the population (also called the **carrying capacity**). Clearly, when $P$ is small compared to $M$, the equation reduces to the exponential one. In order to solve this equation we recognize a nonlinear equation, which is separable. The constant solutions are $P = 0$ and $P = M$. The non-constant solutions may obtained by separating the variables.

$$\frac{dP}{P\left(1 - \frac{P}{M}\right)} = kdt$$

and integration

$$\int \frac{dP}{P\left(1 - \frac{P}{M}\right)} = \int kdt$$

The partial fraction techniques gives

$$\int \frac{dP}{P\left(1-\frac{P}{M}\right)} = \int\left(\frac{1}{P}+\frac{1/M}{1-P/M}\right)dP$$

which gives

$$\ln|P| - \ln\left|1-\frac{P}{M}\right| = kt + c$$

Easy algebraic manipulations give

$$\frac{P}{1-P/M} = Ce^{kt}$$

where $C$ is a constant. Solving for $P$, we get

$$P = \frac{MCe^{kt}}{M + Ce^{kt}}$$

If we consider the initial condition $P(0) = P_0$ (assuming that $P_0$ is not equal to both 0 or $M$), we get

$$C = \frac{P_0 M}{M - P_0}$$

which, once substituted into the expression for $P(t)$ and simplified, we find

$$P(t) = \frac{MP_0}{P_0 + (M - P_0)e^{-kt}}$$

It is easy to see that

$$\lim_{t\to+\infty} P(t) = M$$

However, this is still not satisfactory because this model does not tell us when a population is facing extinction since it never implies that. Even starting with a small population it will always tend to the carrying capacity $M$.

## *3.3 Partial Differential Equations*

Like Ordinary Differential Equations, Partial Differential Equations are equations to be solved in which the unknown element is a function, but in PDEs the function is one of several variables, and so of course the known information relates the function and its partial derivatives with respect to the several variables. Again, one generally looks for qualitative statements about the solution. For example, in many cases, solutions exist only if some of the parameters lie in a specific set (say, the set of integers). Various broad families of PDE's admit general statements about the behaviour of their solutions. This area has a long-standing close relationship with the physical sciences, especially physics, thermodynamics, and quantum mechanics, for many of the topics in the field, the origins of the problem and the qualitative nature of the solutions are best understood by describing the corresponding result in physics.

Roughly corresponding to the initial values in an ODE problem, PDEs are usually solved in the presence of *boundary conditions*. For example, the Dirichlet problem (actually introduced by Riemann) asks for the solution of the Laplace condition on an open subset D of the plane, with the added condition that the value of u on the boundary of D was to be some prescribed function f. (Physically this corresponds to asking, for example, for the steady-state distribution of electrical charge within D when prescribed voltages are applied around the boundary.) It is a nontrivial task to determine how much boundary information is appropriate for a given PDE.

Linear differential equations occur perhaps most frequently in applications (in settings in which a superposition principle is appropriate.) When these differential equations are first-order, they share many features with ordinary differential equations. (More precisely, they correspond to *families* of ODEs, in which considerable attention must be focused on the dependence of the solutions on the parameters.)

Historically, three equations were of fundamental interest and exhibit distinctive behaviour. These led to the clarification of three types of second-order linear differential equations of great interest. The Laplace equation

$$\frac{d^2u}{dx^2}+\frac{d^2u}{dy^2}=0$$

applies to potential energy functions u=u(x,y) for a conservative force field in the plane. PDEs of this type are called ***elliptic***. The Heat Equation

$$\frac{d^2u}{dx^2}+\frac{d^2u}{dy^2}=\frac{du}{dt}$$

applies to the temperature distribution u(x,y) in the plane when heat is allowed to flow from warm areas to cool ones. PDEs of this type are ***parabolic***. The Wave Equation

$$\frac{d^2u}{dx^2}+\frac{d^2u}{dy^2}=\frac{d^2u}{dt^2}$$

applies to the heights u(x,y) of vibrating membranes and other wave functions. PDEs of this type are called ***hyperbolic***. The analyses of these three types of equations are quite distinct in character. Allowing non-constant coefficients, we see that the solution of a general second-order linear PDE may change character from point to point. These behaviours generalize to nonlinear PDEs as well.

A general linear PDE may be viewed as seeking the kernel of a linear map defined between appropriate function spaces. (Determining which function space is best suited to the problem is itself a nontrivial problem and requires careful functional analysis as well as a consideration of the origin of the equation. Indeed, it is the analysis of PDEs which tends to drive the development of classes of topological vector spaces.) The perspective of differential operators allows the use of general tools from linear algebra, including eigenspace decomposition (spectral theory) and index theory.

Modern approaches seek methods applicable to non-linear PDEs as well as linear ones. In this context existence and uniqueness results, and theorems concerning the regularity of solutions, are more difficult. Since it is unlikely that explicit solutions can be obtained for any but the most special of problems, methods of "solving" the PDEs involve analysis within the appropriate function space, for example, seeking convergence of a sequence of functions which can be shown to approximately solve the PDE, or describing the sought for function as a fixed point under a self-map on the function space, or as the point at which some real-valued function is minimized. Some of these approaches may be modified to give algorithms for estimating numerical solutions to a PDE.

Generalizations of results about partial differential equations often lead to statements about function spaces and other topological vector spaces. For example, integral techniques (solving a differential equation by computing a convolution, say) lead to integral operators (transforms on functions spaces); these and differential operators lead in turn to general pseudo differential operators on function spaces.

## 3.3.1 Applications and Related Fields of PDE

1. Differential geometry
2. Global analysis, analysis on manifolds
3. Probability theory and stochastic processes
4. Numerical analysis
5. Mechanics of solids
6. Fluid mechanics
7. Optics, electromagnetic theory
8. Classical thermodynamics, heat transfer
9. Quantum Theory
10. Statistical mechanics, structure of matter
11. Relativity and gravitational theory
12. Geophysics
13. Biology and other natural sciences
14. Systems theory; control

15. Presentation of concepts

## 3.4 Combined Discrete/Continuous Simulation

So far, we saw discrete simulations. This is useful when our problem is like a queue of events, sorted by the simulation time at which they should occur. While Continuous simulation concerns the modeling over time of a system by a representation in which state variables change continuously with respect to time. Typically, we use differential equations that give relationships for the rates of change of the state variables with time.
Discrete simulation is commonly implemented by setting up some sort of queue of operations to be performed. An operation can insert other operations into the queue for immediate execution (appropriate when the consequences of changes to output values must be evaluated immediately), or it can schedule operations for execution at some later time. Systems for discrete simulation are most naturally described in imperative terms , for example

*Operation 23 :*
*When activated :*
*Do certain things;*
*If some condition schedule operation 17 after 0 ticks;*
*Schedule operation 3 after 25 ticks;*
*Schedule operation 23 after 10 ticks.*

Continuous simulation normally requires that each operation be performed at every “tick” of a system clock. While it is possible that some operations may be unnecessary in some cycles because their inputs have not changed, this is an exceptional case. Systems for continuous simulation are most naturally described in declarative terms :

$dm/dt = 2LA - k_2mA - 2k_3m^2 - k_4mr$
$dr/dt = k_2mA - k_4mr - 2k_5r^2$

Our choice of description is not completely determined by the system under investigation alone, in making the choice, we also take into account the amount of detail we wish to reproduce in the simulation. The example given above for continuous simulation is a model of the behaviour of a certain chemical reaction system, at a very fine level of detail, the same system could be described as a set of colliding, and sometimes reacting, molecules, when a discrete simulation would be more natural.
(The choice of implementation technique is even less dependent on the system. Any continuous model, if evaluated on a conventional digital computer, is in fact evaluated discretely, while, if we choose to inspect every operation of a discrete simulation during every computing cycle, it is in effect being evaluated continuously.) If we intend to provide a programme which can simulate anything whatever, we have to provide means of describing both of these forms, both separately ( for systems which are unambiguously of one type or the other ), and together ( for systems with properties naturally handled in both ways ). Perhaps it would be easier if we had a model which would help us to imagine the relationships between the different parts of the system.

*A Working Model*

We live in a world of events, which have unfolding consequences. If we put a ball onto a sloping table, it will begin to roll. To simulate that system, it would be natural to think of the ball's arrival at the table as an event, and to handle the subsequent motion by continuous techniques. But there is a sense in which the continuous part of the behaviour " rolling along the table " is "always there", whenever a ball appears on the table, it will roll subject to the same laws and constraints.
Can we generalize this view? Can we say that the description of a system which we expect to simulate continuously is analogous to the "natural law" of our little universe, which by its nature always applies and must therefore be taken into account continuously, while the events are means of resetting the universe to a new configuration, and, once accomplished, can be forgotten?
Consider the example again. What happens when the ball reaches the edge of the table? We want to switch from the old "natural law" appropriate to rolling along a table to a new one appropriate to falling through the air. Clearly the arrival at the edge of the table must constitute an event. Is it reasonable to describe the effect of the event as transferring the action from one universe (the top of the table) to another (the space between table and floor) ? If we do, we move towards a picture of a problem space split up into a set of disjoint universes, each having its own set of natural laws. If a universe is not at equilibrium, it needs continuous attention; but events can disturb the state of equilibrium of universes (where "disturb" may include "restore", consider the table-top universe after the ball has rolled over the edge).
**A description of the ball and table system :**

*IN UNIVERSE A :*
*Laws :*
*Equations of motion for rolling along the table.*
*If coordinates at edge of table, Schedule Event A2 now.*
*Event A1 :*
*Add a ball at coordinates x, y;*
*Add Universe A to the "continuous attention" set.*
*Event A2 :*
*Remove the ball;*
*Remove Universe A from the "continuous attention" set;*
*Schedule Event B1 now.*
*IN UNIVERSE B :*
Laws :
Equations of motion for falling.
If coordinates at floor …
Event B1 :
Add a ball at coordinates x, y, z;
Add Universe B to the "continuous attention" set.

In the chemical system of the continuous example, m and r represent concentrations of two chemical substances. The point of interest in the system is the behaviour of these concentrations as the light intensity (L) is switched on and off at different frequencies. The complete system can be described in terms something like this:

*Laws :*
$dm/dt = 2LA - k_2mA - 2k_3m^2 - k_4mr;$
$dr/dt = k_2mA - k_4mr - 2k_5r^2.$
*Event 1 :*
*Set L to Lmax.*
*Schedule Event 2 after time ton.*
*Event 2 :*
*Set L to 0.*
*Schedule Event 1 after time toff.*

Thus combined applications (such as Extend) can model systems either discretely or continuously. These hybrid applications combine all the features of both types of modeling. Some systems, especially when a portion of the flow has a delay or wait time, can be modeled as either discrete event or continuous. In this case, you choose how to model the system based on the level of detail required. In general discrete event models provide much more detail about the workings of a system than continuous models do.

## 3.5 The Uses of Simulation in Education

The use of simulated activities in education is widely becoming recognized as an important tool in schools. Schools are finding that activities that promote learning tend to meet the following criteria.

A. They are "real" or virtually real. They simulate some activity so well that real learning takes place. In fact, the term "virtual reality" is now a widely recognized term and one whose implications are important to education. Howard Rheingold's book *Virtual Reality* deals with the technology that "...creates the completely convincing illusion that that one is immersed in a world that exists only inside a computer." Rheingold details his tour through countless situations in which virtual reality is being explored. Educators are not known for having access to state of the art educational technology, but the principles of virtual reality, applied appropriately, are within the grasp of most educators who are serious about the work they do. Using the principles of virtual reality doesn't have to involve the headpieces and the 3-D glasses described by Rheingold, but the concept of simulating reality far educational purposes is an important one.

B They are "hands-on" so that students become participants, not just listeners or observers.

C. They are motivators. Student involvement in the activity is so great that interest in learning more about the activity or the subject, matter of the activity develops.

D. They are age appropriate. Since simulations are designed, they can take into consideration developmental age requirements.

E. They are inspirational. Student input is welcome and activities are designed to encourage students to enhance the activity through their own ideas.

F. They are developmentally valid. Simulations take into account the developmental level of the students.

G. They are empowering. Students take on responsible roles, find ways to succeed, and develop problem- solving tools as a result of the nativity.

The use of simulations puts the teacher into a new role -- a role that is the inevitable result of the evolution of the role of the teacher in education. Most teachers recognize that their role is no longer that of a presenter of information and that students are no longer sponges for facts.

## 3.6 Summary

Continuous simulations are analogous to a constant stream of fluid passing through a pipe. The volume may increase or decrease, but the flow is continuous. In continuous models, values change based directly on changes in time. These values reflect the state of the modeled system at any particular time, and simulated time advances evenly from one time-step to the next. For example, an airplane flying on autopilot represents a continuous system since changes in state (such as position or velocity) change continuously with respect to time. The time line for a continuous model is evenly spaced. Ordinary and partially differential equation are extremely useful to model the behavior of the continuous system. Complex differential equation whether ordinary or partial are generally solved by using numerical techniques using digital computers. In discrete event models, discrete entities change state as events occur in the simulation. Orders arriving, parts being assembled, and customers calling are examples of discrete events. The state of the model changes only when those events occur; the mere passing of time has no direct effect. We can combine continuous and discrete event simulation and this combination some times is called hybrid system simulation. Simulation is extremely helpful in education and training as a learning tool where real learning environment is either costly or dangerous to life, like fighter plane training.

## 3.7 Keywords

Modelling, discrete event simulation, continuous simulation, verification, validation, system, Ordinary Differential Equation (ODE), Partial Differential Equation (PDE), Initial Value Problem (IVP), Virtual Reality, Population Model.

## 3.8 Self Assessment Questions

Q.1 A tank holds 1000 liters of water, in which 15 kg of salt is dissolved. Pure water enters the tank at the rate of 10 liters per minute. The solution is kept thoroughly mixed and is drained from the tank at the same rate. If m is the mass of salt in the tank at time t, which of the following options describes the rate of change of the mass of salt in the tank?

a. $\frac{dm}{dt} = 15 - \frac{m}{100}$ b. $\frac{dm}{dt} = -\frac{m}{100}$

c. $\frac{dm}{dt} = -\frac{15}{100}$ d. $\frac{dm}{dt} = \frac{15-m}{1000}$

Q.2 A tank holds 1000 liters of pure water. Brine which contains 0.05 kg of salt per liter enters the tank at the rate of 5 liters per minute. The solution is kept thoroughly mixed and is drained from the tank at the rate of 5 liters per minute. If m is the mass of salt in the tank at time t, which of the following options describes the rate of change of the mass of salt in the tank?

a. $\frac{dm}{dt} = -\frac{m}{200}$ b. $\frac{dm}{dt} = 0.25$

c. $\frac{dm}{dt} = \frac{50-m}{200}$ d. $\frac{dm}{dt} = 0.05 - \frac{m}{200}$

Q.3 Let $P(t)$ is the population of a certain animal species. Assume that $P(t)$ satisfies the logistic growth equation

$$\frac{dP}{dt} = 0.2P(t)\left(1 - \frac{P(t)}{200}\right), y(0) = 150$$

**1.** Is the above differential equation separable?
**2.** Is the differential equation autonomous?
**3.** Is the differential equation linear?
**4.** Without solving the differential equation, give a sketch of the graph of $P(t)$.
**5.** What is the long-term behavior of the population P(t)?
**6.** Show that the solution is of the form

$$P(t) = \frac{e^{0.2t}}{A + Be^{0.2t}}$$

Find $A$ and $B$.
Hint: Use the initial condition and the result of 5.
**7.** Where is the solution's inflection point?
Hint: This can be done without using the answer of 6.

**8.** What is special about the growth rate of the population $P(t)$ at the inflection point (found in 7)?

Q.4 The fox squirrel is a small mammal native to the Rocky Mountains. These squirrels are very territorial. Note the following observations:
if the population is large, their rate of growth decreases or even becomes negative;
if the population is too small, fertile adults run the risk of not being able to find suitable mates, so again the rate of growth is negative
The carrying capacity $N$ indicates when the population is too big, and the sparsity parameter $M$ indicates when the population is too small. A mathematical model, which agrees with the above assumptions, is the modified logistic model

$$\frac{dP}{dt} = kP\left(1-\frac{P}{N}\right)\left(\frac{P}{N}-1\right)$$

**1.** Find the equilibrium (critical) points. Classify them as, source, sink or node. Justify your answers.
**2.** Sketch the slope-field.
**3.** Assume N=100 and M=1 and k = 1. Sketch the graph of the solution which satisfies the initial condition $y(0)=20$.
**4.** Assume that squirrels are emigrating (from a certain region) with a fixed rate $E$. Write down the new differential equation.
Also, discuss the equilibrium (critical) points under the parameter $E$. When do you observe a bifurcation?

## 3.9 Reference /Suggested Reading


*11.* G.Gorden, "System Simulation",PHI.
*12.* Erwin Kreyszig," Advanced Engineering Mathematics", Wiley

**Subject : System Simulation and Modeling Author : Jagat Kumar**
**Paper Code: MCA 504 Vetter : Prof. Dharminder Kumar**
**Lesson : Discrete System Simulation**
**Lesson No. : 04**

**Structure**

## 4.0 Objective

As we studied in previous few units that simulation is a powerful technique for solving wide variety of problems. In this unit we will studied a discrete event simulation in more detail. As we learn in unit I that the model used in discrete system simulation has a set of numbers to represent the state of the system is called a state descriptor. We will also learn about queuing simulation, which is very important aspect in discreet event simulation along with simulation of time-sharing system.

## 4.1. Introduction

Simulation involves the development of descriptive computer models of a system and exercising those models to predict the operational performance of the underlying system being modeled. Systems that change with time, such as a gas station where cars come and go (called dynamic systems) and involve randomness. Nobody can guess at exactly which time the next car should arrive at the station, are good candidates for simulation. Modeling complex dynamic systems theoretically need too many simplifications and the emerging models may not be therefore valid.
Suppose we are interested in a gas station. We may describe the behavior of this system graphically by plotting the number of cars in the station; the state of the system. Every time a car arrives the graph increases by one unit while a departing car causes the graph to drop one unit. This graph (called sample path), could be obtained from observation of a real station, but could also be artificially constructed. Such artificial construction and the analysis of the resulting sample path (or more sample paths in more complex cases) consist of the simulation.

## 4.2 Time Graph Representation

Every System based on the change of time. So in a system model time counting is a crucial thing. In a graph time is recorded by a number called clock time or time counter. Initially it is set on zero. Two basic methods exits for updating clock time.

**1. Time Slicing:** Advances the model by a fixed amount each time, regardless of the absence of any events to carry out.

**2. Next Event:** Advances the model to the next event to be executed, regardless of the time interval. This method is more efficient than Time Slicing, especially where events

are infrequent, but can be confusing when being represented graphically (processes that take different times will appear to happen in the same time frame if the stop event is the next event after the start event).
The first method is called “Interval-oriented” and another one is called “Event-oriented”.
Since the early 1960's, Simulation has been one of many methods used to aid strategic decision-making within industry. Its main strength lies in the ability to imitate complex real world problems and to analyze the behaviour of the system as time progresses.

## 4.3 Discrete Simulation

We already learned in unit I, in the classification of simulation, that simulation can be of

- Continuous Simulation and
- Discrete Simulation.

What is Discrete Simulation? The Oxford English Dictionary describes Discrete Simulation as:

> "The technique of imitating the behaviour of some situation or system (Economic, Mechanical etc.) by means of an analogous model, situation, or apparatus, either to gain information more conveniently or to train personnel."

In some systems the state changes all the time, not just at the time of some discrete events. For example, the water level in a reservoir with given in and outflows may change all the time. In such cases "continuous simulation" is more appropriate, although discrete event simulation can serve as an approximation. When the number of events is finite, we call the simulation "discrete event."
Discrete event Simulation is a powerful computing technique for understanding the behavior of systems. The particular nature of the system and the properties we wish to understand can vary. Here are three examples:

1. A natural scientist may be interested in a system of wolves and sheep, where the number of wolves changes with a constant birth rate and a death rate that is inversely proportional to the number of sheep, and the number of sheep changes with a constant birth rate and a death rate that is directly proportional to the number of wolves. The scientist would like to know the following: Do the number of wolves and the number of sheep stabilize in the long run, and if so to what values? Or do they vary cyclically, and if so with what period and phase?
2. A computer scientist may be interested in a system of jobs that circulate in a network of servers (e.g., CPU’s and I/O devices). The computer scientist would like to know whether a particular server is a ‘‘bottleneck’’, i.e., in the long run, is that server always busy while the other servers are mostly idle.
3. A classical system example is a queuing system with a single server. Here, customers arrive with certain service requirements, get served in some order, say first-come-first-served, and depart when their service is completed. Note that a customer who arrives when the server is busy has to wait (in a queue). For this system, we would like to determine the average waiting time for customers, the average number of customers in the system, the fraction of time the server is busy, etc.

Simulations may be performed manually. Most often, however, the system model is written either as a computer program as some kind of input into simulator software. The Figure 1 below shows the key stages in using Discrete Event Simulation. It can be noted that this bears a strong resemblance to other simulation techniques and other analysis program development methodologies

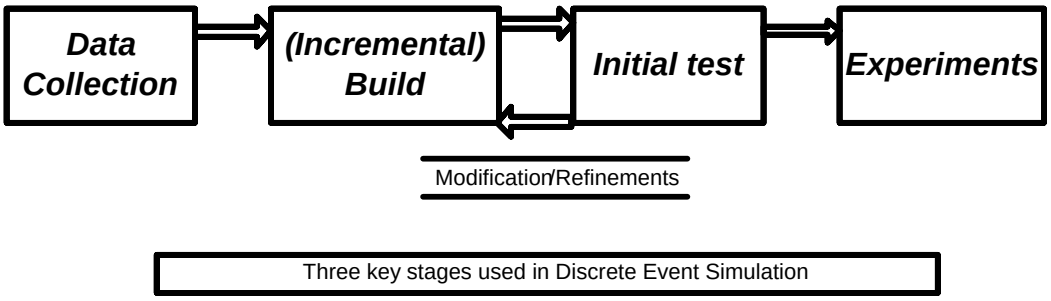



**Figure 1:** Key Stages in Using Discrete Event Simulation

## 4.3.1 Key Principles

Although, discrete event simulation could conceivably be carried out by hand it can be computationally intensive, therefore will invariably involve computers and software. The software could be a high level programming language such as Pascal, C++ or any specialized simulation language. The five key features found in the software simulation model are:

**1. Entities:** Representations of real-life elements e.g. in manufacturing these could be parts or machines.

**2. Relationships:** Link entities together e.g. a part may be processed by a machine.

**3. Simulation Executive:** Responsible for controlling the time advance and executing discrete events.

**4. Random Number Generator:** Helps to simulate different data coming into the simulation model. Important that the random data can be reproduced in different simulation runs.

**5. Results & Statistics:** Important in validating the model and for providing performance measures.

There are also three approaches to describing the discrete simulation, see the Figure 2 .

**1. Event:** This approach describes an instantaneous change, usually from a stop event to a start event. This is the most common one used, easy to understand and efficient and is acceptable to implement.

**2. Activities:** Represents duration. Essentially groups a number of events in order to describe an activity carried out by an entity e.g. a machine loading. This approach is easy to understand and to implement but is not efficient.

**3. Process:** This approach groups activities to describe the life cycle of an entity e.g. a machine. This is less common and more difficult to plan and implement, but is generally thought to be the most efficient.

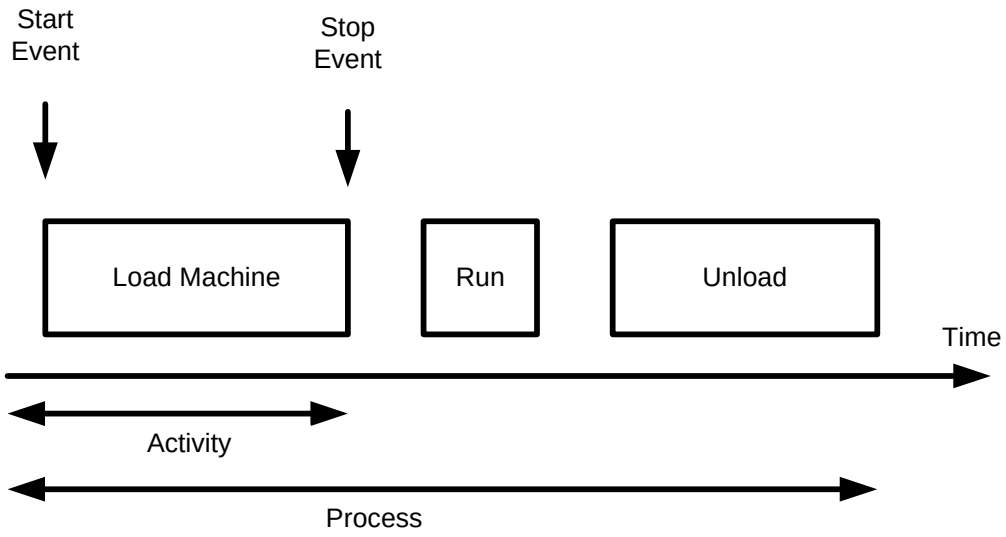



**Figure 2**: Three Approaches to Describing the Discrete Simulation

**A Simple Example:** Building a simulation of gas station with a single pump served by a single service man. Assume that arrival of cars as well their service times are random. At first identify the:

1. **states:** number of cars waiting for service and number of cars served at any moment
2. **events:** arrival of cars, start of service, end of service
3. **entities:** these are the cars
4. **queue:** the queue of cars in front of the pump, waiting for service
5. **random realizations:** inter-arrival times, service times
6. **distributions:** we shall assume exponential distributions for both the inter-arrival time and service time.

Next, specify what to do at each event. The above example would look like this: At event of entity arrival: Create next arrival. If the server is free, send entity for start of service. Otherwise it joins the queue. At event of service start: Server becomes occupied. Schedule end of service for this entity. At event of service end: Server becomes free. If any entities waiting in queue: remove first entity from the queue; send it for start of service. Some initiation is still required, for example, the creation of the first arrival. Lastly, the above is translated into code. This is easy with an appropriate library, which has subroutines for creation, scheduling, and proper timing of events, queue manipulations, random variable generation and statistics collection.

Discrete simulation is a technique where the simulation is advanced from event time to event time rather than using a continuously advancing time clock as in continuous simulation. Suppose we consider the example of the interaction of the principle and interest associated with a savings account. This can be represented by a systems thinking diagram, Figure 3, as follows:

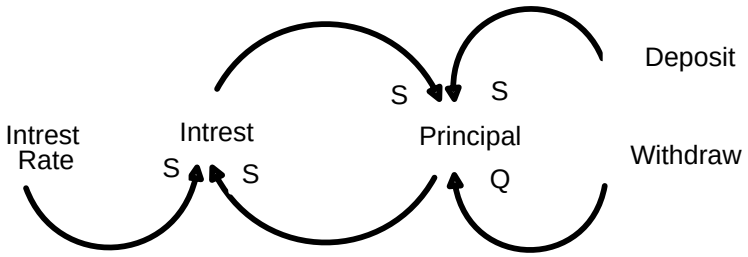



**Figure 3: Interaction of the Principle and Interest**

This diagram indicates that **Deposits** increase the **Principal** and **Withdraws** decrease the **Principal**. Also, the **Principal** interacts with the **Interest Rate** on some periodic basis to create **Interest**. The **Interest** then serves to increase the **Principal**. If we then turn this into a 10-year simulation with the assumptions that the **Principal** is initially 100, there are no **Deposits** or **Withdraws**, and an **Interest Rate** of 5% is paid once a year it might look like this in Extend.
If we draw the graph between principal and time (year) of running the model for 10 years. Note that the **Interest** is computed and added to the **Principal** at the end of each year and the discrete nature of the simulation is very evident.

## 4.4 Simulation of Queuing System

A queue is waiting line for service i.e the combination of all entities in the system-those being served, and those waiting for service-will be called a queue. People, cars, trucks, ships, T.V's, arrive at a certain place to be serviced in some way. The important parameters in a queuing system are:

1. The arrival pattern of customers
2. The service pattern
3. The no. of servers
4. The queue discipline

### *4.4.1 The Single-Server Queue*

The simplest queuing system is depicted in Figure 4. The central element of the system is a server, which provides some service to items. Items from some population of items

arrive at the system to be served. If the server is idle, an item is served immediately. Otherwise, an arriving item joins a waiting line (The waiting line is referred to as a queue in some treatments in the literature; it is also common to refer to the entire system as a queue. Unless otherwise noted, we use the term *queue* to mean waiting line). When the server has completed serving an item, the item departs. If there are items waiting in the queue, one is immediately dispatched to the server. The server in this model can represent anything that performs some function or service for a collection of items. Examples: a processor provides service to processes; a transmission line provides a transmission service to packets or frames of data; an I/O device provides a read or writes service for I/O requests. The Single-Server Queue is described in more detail in section 4.5 of this unit.

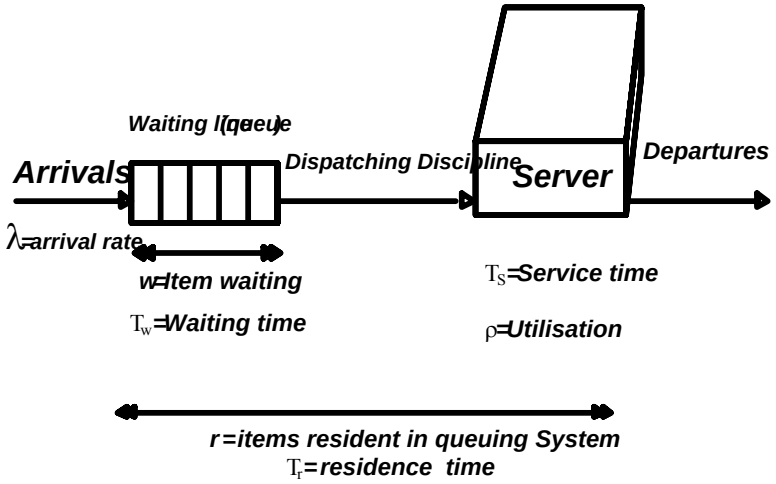



**Figure: 4 Queuing System Structure and Parameters for Single-server Queue**

## 4.4.2Queue Parameters

Figure 4 also illustrates some important parameters associated with a queuing model. Items arrive at the facility at some average rate (items arriving per second) λ. At any given time, a certain number of items will be waiting in the queue (zero or more); the average number waiting is *w*, and the mean time that an item must wait is *Tw*. *Tw* is averaged over all incoming items, including those that do not wait at all. The server handles incoming items with an average service time *Ts*; this is the time interval between the dispatching of an item to the server and the departure of that item from the server. Utilization, ρ, is the fraction of time that the server is busy, measured over some interval of time. Finally, two parameters apply to the system as a whole. The average number of items resident in the system, including the item being served (if any) and the items waiting (if any), is *r*; and the average time that an item spends in the system, waiting and being served, is *Tr*; we refer to this as the mean residence time.

If we assume that the capacity of the queue is infinite, then no items are ever lost from the system; they are just delayed until they can be served. Under these circumstances, the departure rate equals the arrival rate. As the arrival rate, which is the rate of traffic passing through the system, increases, and the utilization increases and with it,

congestion. The queue becomes longer, increasing waiting time. At $\rho = 1$, the server becomes saturated, working 100% of the time. Thus, the theoretical maximum input rate that can be handled by the system is:

$$\lambda_{max} = \frac{1}{T_S}$$

However, queues become very large near system saturation, growing without bound when $\rho = 1$. Practical considerations, such as response time requirements or buffer sizes, usually limit the input rate for a single server to 70-90% of the theoretical maximum. To proceed, we need to make some assumption about this model:

- **Item population:** Typically, we assume an infinite population. This means that the arrival rate is not altered by the loss of population. If the population is finite, then the number of items reduces the population available for arrival currently in the system; this would typically reduce the arrival rate proportionally.
- **Queue size:** Typically, we assume an infinite queue size. Thus, the waiting line can grow without bound. With a finite queue, it is possible for items to be lost from the system. In practice, any queue is finite. In many cases, this will make no substantive difference to the analysis. We address this issue briefly, below.
- **Dispatching discipline:** When the server becomes free, and if there is more than one item waiting, a decision must be made as to which item to dispatch next. The simplest approach is first in, first out; this discipline is what is normally implied when the term *queue* is used. Another possibility is last in, first out. One that you might encounter in practice is a dispatching discipline based on service time. For example, a packet-switching node may choose to dispatch packets on the basis of shortest first (to generate the most outgoing packets) or longest first (to minimize processing time relative to transmission time). Unfortunately, a discipline based on service time is very difficult to model analytically.

Table 1 and Table 2 summarizes the notation that is used in Figure 4 and introduces some other parameters that are useful. In particular, we are often interested in the variability of various parameters, and this is neatly captured in the standard deviation.

**Table 1 Notation for Queuing Systems**

| | |
|---|---|
| $\lambda$ | = arrival rate; mean number of arrivals per second |
| $T_s$ | = mean service time for each arrival; amount of time being served, not counting time waiting in the queue |
| $\sigma_{Ts}$ | = standard deviation of service time |
| $\rho$ | = utilization; fraction of time facility (server or servers) is busy |
| u | = traffic intensity |
| r | = mean number of items in system, waiting and being served (residence time) |

| | |
|---|---|
| R | = number of items in system, waiting and being served |
| $T_r$ | = mean time an item spends in system (residence time) |
| $T_R$ | = time an item spends in system (residence time) |
| $\sigma_r$ | = standard deviation of r |
| $\sigma_{Tr}$ | = standard deviation of $T_r$ |
| w | = mean number of items waiting to be served |
| $\sigma_w$ | = standard deviation of w |
| $T_w$ | = mean waiting time (including items that have to wait and items with waiting time = 0) |
| $T_d$ | = mean waiting time for items that have to wait |
| N | = number of servers |
| $m_x(y)$ | = the yth percentile; that value of y below which x occurs y percent of the time |

**Table 2 Some Basic Queuing Relationships**

| General | Single Server | Multi-server |
|---|---|---|
| $r = \lambda T_r$ Little's formula<br>$w = \lambda T_w$ Little's formula<br>$T_r = T_w + T_S$ | $\rho = \lambda T_S$<br>$r = w + \rho$ | $\rho = \frac{\lambda T_S}{N}$<br>$u = \lambda T_S = \rho N$<br>$r = w + N\rho$ |

### 4.4.3 The Multi-Server Queue

Figure 5 shows a generalization of the simple model we have been discussing for multiple servers, all sharing a common queue. If an item arrives and at least one server is available, then the item is immediately dispatched to that server. It is assumed that all servers are identical; thus, if more than one server is available, it makes no difference which server is chosen for the item. If all servers are busy, a queue begins to form. As soon as one server becomes free, an item is dispatched from the queue using the dispatching discipline in force.

With the exception of utilization, all of the parameters illustrated in Figure 4 carry over to the multiserver case with the same interpretation. If we have *N* identical servers, then ρ is the utilization of each server, and we can consider $N_\rho$ to be the utilization of the entire system; this latter term is often referred to as the traffic intensity, *u*. Thus, the theoretical maximum utilization is *N* *100%, and the theoretical maximum input rate is:

$$\lambda_{max} = \frac{N}{T_S}$$

The key characteristics typically chosen for the multi-server queue correspond to those for the single-server queue. That is, we assume an infinite population and an infinite

queue size, with a single infinite queue shared among all servers. Unless otherwise stated, the dispatching discipline is FIFO. For the multi-server case, if all servers are assumed identical, the selection of a particular server for a waiting item has no effect on service time.

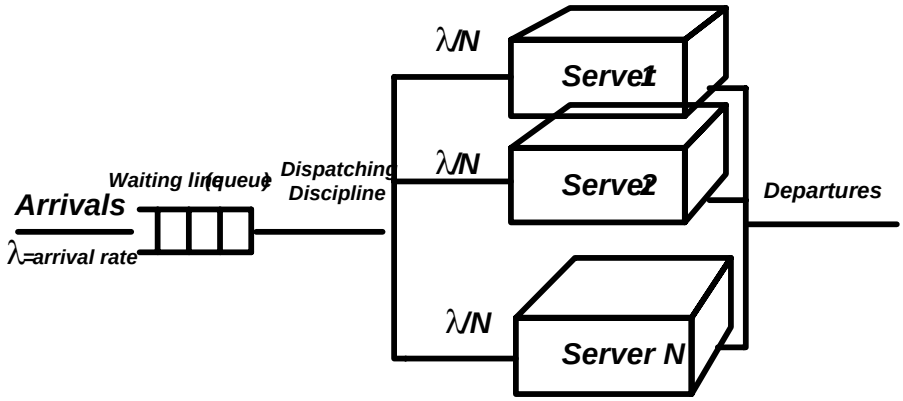



**Figure 5: Multi-server queue**

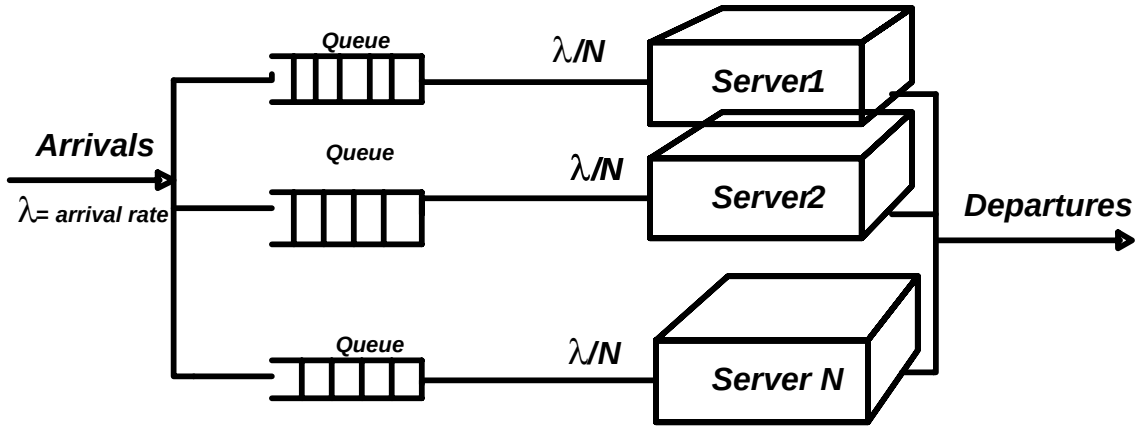



**Figure 6: Multi server Versus Multiple Single-server Queues**

By way of contrast, Figure 6 shows the structure of multiple single-server queues. As we shall see, this apparently minor change in structure has a significant impact on performance. The Multi-Server Queue is described in more detail in section 4.6 of this unit.

## 4.4.4 Basic Queuing Relationships

To proceed much further, we are going to have to make some simplifying assumptions. These assumptions risk making the models less valid for various real-world situations. Fortunately, in most cases, the results will be sufficiently accurate for planning and design purposes. There are, however, some relationships that are true in the general case,

and these are illustrated in Table 2. By themselves, these relationships are not particularly helpful.

## Assumptions

The fundamental task of a queuing analysis is as follows: Given the following information as input:

- Arrival rate
- Service time

Provide as output information concerning:

- Items waiting
- Waiting time
- Items in residence
- Residence time.

What specifically would we like to know about these outputs? Certainly we would like to know their average values ($w$, $Tw$, $r$, $Tr$). In addition, it would be useful to know something about their variability. Thus, the standard deviation of each would be useful $\left(\sigma_w, \sigma_{T_W}, \sigma_r, \sigma_{T_r}\right)$. Other measures may also be useful. For example, to design a buffer associated with a bridge or multiplexer, it might be useful to know for what buffer size the probability of overflow is less than 0.001. That is, what is the value of $M$ such that Pr [items waiting < $M$] = 0.999?

To answer such questions in general requires complete knowledge of the probability distribution of the arrival rate and service time. Furthermore, even with that knowledge, the resulting formulas are exceedingly complex. Thus, to make the problem tractable, we need to make some simplifying assumptions.

The most important of these assumptions is that the arrival rate obeys the Poisson distribution, which is equivalent to saying that the inter arrival times are exponential,

which is equivalent to saying that the arrivals occur randomly and independent of one

another. This assumption is almost invariably made. Without it, most queuing analysis is

impractical. With this assumption, it turns out that many useful results can be obtained if

only the mean and standard deviation of the arrival rate and service time are known.

Matters can be made even simpler and more detailed results can be obtained if it is

assumed that the service time is exponential or constant.

A convenient notation has been developed for summarizing the principal assumptions that are made in developing a queuing model. The notation is X/Y/N, where X refers to the distribution of the inter-arrival times, Y refers to the distribution of service times, and N refers to the number of servers. The most common distributions are denoted as follows:

G = general independent arrivals or service times
M = negative exponential distribution
D = deterministic arrivals or fixed length service.

Thus, M/M/1 refers to a single-server queuing model with Poisson arrivals and exponential service times.

## 4.5 SINGLE-SERVER QUEUES

Table 3 (column a) provides some equations for single server queues that follow the M/G/1 model. That is, the arrival rate is Poisson and the service time is general. Making use of a scaling factor, *A*, the equations for some of the key output variables is straightforward. Note that the key factor in the scaling parameter is the ratio of the standard deviation of service time to the mean. No other information about the service time is needed. Two special cases are of some interest. When the standard deviation is equal to the mean, the service time distribution is exponential (M/M/1). This is the simplest case and the easiest one for calculating results. Table 3 (column b) shows the simplified versions of equations for the standard deviation of *r* and *Tr*, plus some other parameters of interest. The other interesting case is a standard deviation of service time equal to zero, that is, a constant service time (M/D/1). The corresponding equations are shown in Table 3(column c).

Figures 4 and 5 plot values of average queue size and residence time versus utilization for three values of $\sigma_{T_s} / T_s$. This latter quantity is known as the **coefficient of variation**, and gives a normalized measure of variability. Note that the poorest performance is exhibited by the exponential service time, and the best by a constant service time. Usually, one can consider the exponential service time to be a worst case. An analysis based on this assumption will give conservative results. This is nice, because tables are available for the M/M/1 case and values can be looked up quickly.
What value of $\sigma_{T_s} / T_s$ is one likely to encounter? We can consider four regions:

- **Zero:** This is the rare case of constant service time. For example, if all transmitted messages are of the same length, they would fit this category.

- **Ratio less than 1:** Because this ratio is better than the exponential case, using M/M/1 tables will give queue sizes and times that are slightly larger than they should be. Using the M/M/1 model would give answers on the safe side. An example of this category might be a data entry application for a particular form.

- **Ratio close to 1:** This is a common occurrence and corresponds to exponential service time. That is, service times are essentially random. Consider message lengths to a computer terminal: a full screen might be 1920 characters, with message sizes varying over the full range. Airline reservations, file lookups on inquires, shared LAN, and packet-switching networks are examples of systems that often fit this category.

- **Ratio greater then 1:** If you observe this, you need to use the M/G/1 model and not rely on the M/M/1 model. A common occurrence of this is a bimodal distribution, with a wide spread between the peaks. An example is a system that experiences many short messages, many long messages, and few in between.

The same consideration applies to the arrival rate. For a Poisson arrival rate, the inter-arrival times are exponential, and the ratio of standard deviation to mean is 1. If the observed ratio is much less than 1, then arrivals tend to be evenly spaced (not much variability), and the Poisson assumption will overestimate queue sizes and delays. On the other hand, if the ratio is greater than 1, then arrivals tend to cluster and congestion becomes more acute.

**Table 3 Formulas for Single-Server Queues**

Assumptions: 1. Poisson arrival rate.
2. Dispatching discipline does not give preference to items based on service times.
3. Formulas for standard deviation assume first-in, first-out dispatching.
4. No items are discarded from the queue.

| a. General Service Times (M/G/1) | (b) Exponential Service Times (M/M/1) | (c) Constant Service Times (M/D/1) |
|---|---|---|
| $A = \frac{1}{2}\left[1 + \left(\frac{\sigma_{T_S}}{T_S}\right)^2\right]$<br>$r = \rho + \frac{\rho^2 A}{(1-\rho)}$<br>$w = \frac{\rho^2 A}{1-\rho}$<br>$T_r = T_S + \frac{\rho T_S A}{1-\rho}$<br>$T_w = \frac{\rho T_S A}{1-\rho}$ | $r = \frac{\rho}{1-\rho}$ $\quad w = \frac{\rho^2}{1-\rho}$<br>$T_r = \frac{T_S}{1-\rho}$ $\quad T_w = \frac{\rho T_S}{1-\rho}$<br>$\sigma_r = \frac{\sqrt{\rho}}{1-\rho}$ $\quad \sigma_{T_r} = \frac{T_S}{1-\rho}$<br>$\Pr[R = N] = (1-\rho)\rho^N$<br>$\Pr[R <= N] = \sum_{i=0}^{N}(1-\rho)\rho^i$<br>$\Pr[T_R <= T] = 1 - e^{-(1-\rho)t/T_S}$<br>$m_{T_r}(y) = T_r \times \ln\left(\frac{100}{100-y}\right)$<br>$m_{T_r}(y) = \frac{T_w}{\rho} \times \ln\left(\frac{100\rho}{100-y}\right)$ | $r = \frac{\rho^2}{2(1-\rho)} + \rho$<br>$w = \frac{\rho^2}{2(1-\rho)}$<br>$T_r = \frac{T_S(2-\rho)}{2(1-\rho)}$<br>$T_w = \frac{\rho T_S}{2(1-\rho)}$<br>$\sigma_r = \frac{1}{1-\rho}\sqrt{\rho - \frac{3\rho^2}{2} + \frac{5\rho^3}{6} - \frac{\rho^4}{12}}$<br>$\sigma_{T_r} = \frac{T_S}{1-\rho}\sqrt{\frac{\rho}{3} - \frac{\rho^2}{12}}$ |

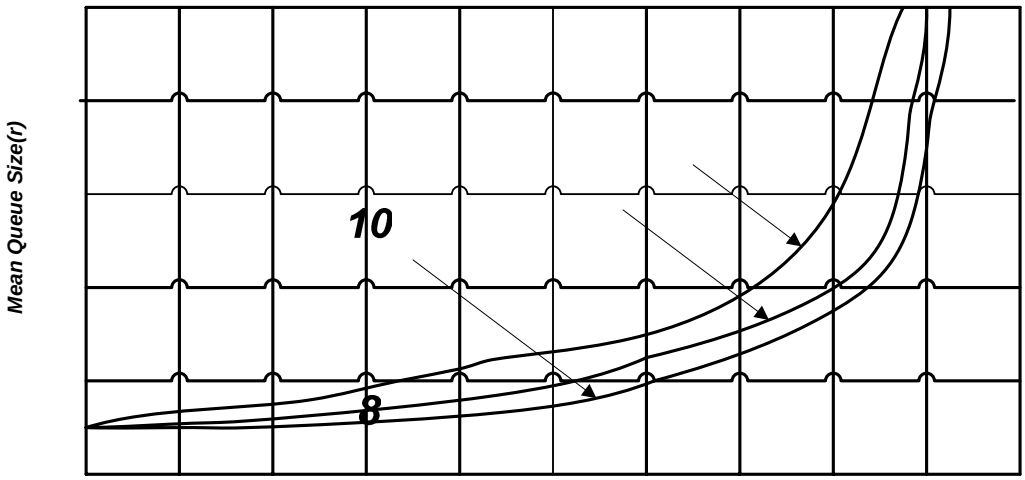



**Figure 7: Mean Queue Size for Single-Server Queue**

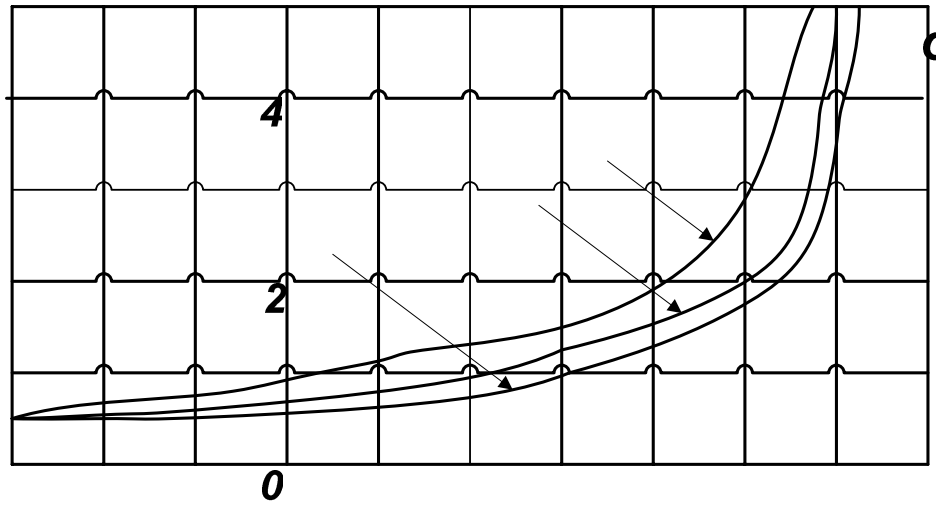



**Figure 8: Mean Residence time for Single-server queue.**

## 4.6 MULTISERVER QUEUES

Table 4 lists formulas for some key parameters for the multi-server case. Note the restrictiveness f the assumptions. Useful congestion statistics for this model have been obtained only for the case of M/M/N, where the exponential service times are identical for the *N* servers.

**Table 4 Formulas for Multi-server Queues (M/M/N)**

Assumptions:

1. Poisson arrival rate.
2. Exponential service times
3. All servers equally loaded
4. All servers have same mean service time
5. First-in, first-out dispatching
6. No items are discarded from the queue

$$K = \frac{\sum_{I=0}^{N-1} \frac{(N\rho)^I}{I!}}{\sum_{I=0}^{N-1} \frac{(N\rho)^I}{I!}}$$ Poisson ratio function

Erlang-C function=Probability that all servers are busy $= C = \frac{1-K}{1-\rho K}$

$$r = C\frac{\rho}{1-\rho} + N\rho \qquad w = C\frac{\rho}{1-\rho}$$

$$T_r = \frac{C}{N}\frac{T_S}{1-\rho} + T_S \qquad T_w = \frac{C}{N}\frac{T_S}{1-\rho}$$

$$\sigma_{T_r} = \frac{T_S}{N(1-\rho)}\sqrt{C(2-C) + N^2(1-\rho)^2}$$

$$\sigma_w = \frac{1}{(1-\rho)}\sqrt{C\rho(1+\rho-C\rho)}$$

$$\Pr[T_W > t] = Ce^{-N(1-\rho)t/T_S}$$

$$m_{T_W}(y) = \frac{T_S}{N(1-\rho)} \times \ln\left(\frac{100C}{100-y}\right)$$

$$T_d = \frac{1}{N}\frac{T_S}{1-\rho}$$

## 4.7 Performance Measures for Queuing Systems

Consider the queuing system with a single server mentioned in section 1.6. Let customer *n* denote the *nth cus*tomer to arrive at the queuing system, for *n* = 1, 2, ....

Let us represent the state of the system by the queue of customers in the system, in the order of their arrival. For example, < 3, 4, 5 > means that customers 3, 4 and 5 are in the system by convention, the head of the queue is at the left. If the queue is not empty, then the customer at the head is being served. We use <> to denote an empty queue.

Let the events of the system be *Arrival*(*n*) denoting the arrival of customer *n*, and *Departure*(*n*) denoting the departure of customer *n*. (This assumes that if a customer completes service when other customers are waiting then the next customer's service is started immediately; otherwise, we would need another event representing the start of service.)

Let *S(n)* denote the service time of customer *n*; i.e., customer *n* requires the server's attention for *S(n) seconds* (Without loss of generality, we assume that time units are seconds.) Let *TA(n)* denote the arrival time of the customer n.

The following sequence represents an evolution of the system, assuming that *S(n)* equals 2.0 seconds for all n, and *TA(n)* equals 2.5*n*- 2. 5 seconds for odd *n* and 2.5*n*-4 seconds for even *n* (i.e., customers arrive at times0. 0, 1. 0, 5. 0, 6. 0, 10. 0, 11. 0, ...). For readability, each element of the evolution is listed on a new line. (Observe that the system evolution is cyclic with a period of 5 seconds.)

| States | Event | Occurrence time |
|---|---|---|
| < > | | |
| | *Arrival*(1) | 0.0 |
| <1> | | |
| | *Arrival*(2) | 1.0 |
| <1, 2> | | |
| | *Departure*(1) | 2.0 |
| <2> | | |
| | *Departure*(2) | 4.0 |
| < > | | |
| | *Arrival*(3) | 5.0 |
| <3> | | |
| | *Arrival*(4) | 6.0 |
| <3, 4> | | |
| | *Departure*(3) | 7.0 |
| <4> | | |
| | • | |
| | • | |
| | • | |

A queuing system has many *performance measures* of interest. We will look at some of them, namely, (1) the average number of customers (also called average system size), (2) the average response time, (3) the average waiting time, and (4) the throughput.

## 4.7.1 Average Number of Customers

Let *N*(*t*) denote the number of customers in the system at time *t*. *N*(*t*) is an integer-valued discontinuous function that increases by 1 at each arrival and decreases by 1 at each departure. The following graph shows *N(t)* versus *t*.


**Fig: 9 Response graph of customer 1 and 2**

For a given evolution, the *average number of customers in the system*, which we shall denote by $N$, is define to be the average of $N(t)$ over time for the evolution. Formally, if the time duration of the evolution is $T$ seconds,
Then

$$N = \frac{1}{T}\int_0^T N(t)dt$$

To illustrate, let us consider the evolution of our queuing system until just after the departure of customer 2. For this evolution, $N$ equals 1.25. (It is obtained as follows. Customer 2 departs at time 4. $\int_0^4 N(t)dt$ (Which is the area under $N(t)$ from time 0 to 4) equals 5. Thus, the average system size is, $\int_4^5 N(t)dt$ (Which is the area under $N(t)$ from time 4 to 5) equals 1.25.)

In general, we want the ‘‘steady-state’’ value of $N$, i.e., $N$ for extremely ‘‘long’’ evolutions. Formally, we want

$$N = Lim_{T\to\infty}\frac{1}{T}\int_0^T N(t)dt$$

In the above queuing example, the steady-state $N$ equals 1.0. (We can obtain it easily by noting that the evolution repeats itself every 5 seconds. Thus, it is sufficient to obtain $N$ for any contiguous 5 second interval, such as [0, 5]. $\int_0^5 N(t)dt$ equals 5. Thus, the average system size is, $\int_5^5 N(t)dt$ (which is the area under $N(t)$ from time 5 to 5) equals 1.0.)

## 4.7.2 Average Response Time

The response time of customer $n$, denoted by $R_n$, is the time spent by the customer in the system. For a given evolution, the *average response time*, which we shall denote by $R$, is the average of the $R_n$*'s for the customers* departing in the evolution. Formally, if $K$ customers depart in the evolution, then

$$\frac{1}{K}\sum_{i=0}^{K} R_i$$

To illustrate, let us consider the evolution of our queuing system until just after the departure of customer 2. For this evolution, $R$ equals 2.5. (It is obtained as follows. There are two departures in this **simulation**, namely customers 1 and 2. The response time of customer 1 is 2.0 seconds. The response time of customer 2 is 3.0 seconds. Thus, the average response time is, $\frac{2.0+3.0}{2}$ which equals 2.5.)

In general, we want the ''steady-state'' $R$, i.e., for extremely ''long'' evolutions. That is, we want

$$R = Lim_{K\to\infty} \frac{1}{K}\sum_{i=0}^{K} R_i$$

For the above queuing example, $R_n$ equals 2.0 seconds for odd $n$ and 3.0 seconds for even $n$. Thus steady-state $R$ equals 2.5 seconds per customer.

## 4.7.3Average Waiting Time

The waiting time of customer $n$, denoted by $R_{n\ n}$, is defined by $S_n = R_n - S_n$. For an evolution, the average waiting time, denoted $W$, is the average of the $W_n$'s for the customers departing in the evolution. (For the above evolution, the steady-state $W$ equals 0.5 seconds per customer.)

## 4.7.4 Throughput

For an evolution, the throughput, denoted by $X$, indicates the number of departures over the total time of the evolution. (For the above evolution, the steady-state $X$ equals 0.4 customers per second.)

## 4.7.5 General Comments

Note that $R$ and $W$ are customer averages, whereas $N$ and $X$ are time averages. In general, when we refer to a performance measure we mean its steady-state value, unless otherwise mentioned. Note that the steady-state averages do not always exist. For example in the above queuing system, if $S_n$ were greater than 2.5, R, W, N would not exist.
The input parameters of the above queuing system are $\{S_n\}$ and $\{TA_n\}$. In the above description, we have described them deterministically. As mentioned in Section 1, we typically want to describe them probabilistically.

For example, instead of having $S_n$ equal 2.0 seconds for all $n$, we may want $S_n$ to be a value between 1.7 to 2.3 seconds, such that each value in the range is chosen with uniform probability and successive values of $S_n$ *are chosen* independently.

Observe that the above values for $N$, $R$ and $X$ satisfy the following:

$$N = R \times X$$

This is not a coincidence. In fact, this is a very important relationship, called ***Little's Law***. It holds for *any* general system in steady state!

## 4.8 The Simulation of Time Sharing Systems

Multiprogrammed batched system provides an environment where various system resources are utilized effectively. Time-sharing, or Multitasking, is logical extension of multiprogramming. Multiple jobs are executed by CPU switching between them, but the switches occur so frequently that the user may interact with each program while it is running.

Time-sharing systems were developed to provide interactive use of a computer system at a reasonable cost. A time sharing system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time shared system. Each user has at least one separate program in memory. A time shared operating system allows the many user to share the computer simultaneously. As the system switches rapidly from one user to the next, each user is given the impression that she has her own computer, whereas actually one computer is being shared among many users. Time-shared operating systems are even more complex then are multiprogrammed operating systems.

Multiprogramming and time sharing system are the central term of modern operating systems. Here we present the one example of simulation of time sharing system.

One such example is the SimOS simulation environment developed by Stanford University SimOS is an environment for studying the hardware and software of computer systems.SimOS simulates the hardware of a computer system in enough detail to boot a commercial operating system and run realistic workloads on top of it. SimOS includes multiple interchangeable simulation models for each hardware component. By selecting the appropriate combination of simulation models, the user can explicitly control the tradeoff between simulation speed and simulation detail. To handle the large amount of low-level data generated by the hardware simulation models, SimOS contains flexible annotation and event classification mechanisms that map the data back to concepts meaningful to the user. SimOS has been extensively used to study new computer hardware designs, to analyze application performance, and to study operating systems. SimOS is a machine simulation environment designed to study large complex computer systems. SimOS differs from most simulation tools in that it simulates the complete hardware of the computer system. SimOS simulates the computer hardware with sufficient speed and detail to run existing system software and application programs. For example, the current version of SimOS simulates the hardware of multiprocessor computer systems in enough detail to boot, run, and study Silicon Graphics' IRIX operating system as well as any application that runs on it, such as parallel compilation and commercial relational database systems. Despite its name, SimOS does not model an

operating system or any application software, but rather models the hardware components of the target machine. SimOS contains software simulation of all the hardware components of modern computer systems: processors, memory management units (MMU), caches, memory systems, as well as I/O devices such as SCSI disks, Ethernets, hardware clocks, and consoles. SimOS currently simulates the hardware of MIPS-based multiprocessors in enough detail to boot and run an essentially unmodified version of commercial operating system, Silicon Graphics' IRIX.

## 4.9 Summary

Discrete event Simulation is a powerful computing technique for understanding the behavior of systems. A *discrete event* is something that occurs at an instant of time. For example, pushing an elevator button, starting of a motor, stopping of a motor, and turning on a light, are all discrete events because there is an instant of time at which each occurs. Activities such as moving a train from point A to point B are not discrete events because they have a time duration. Every System based on the change of time. So discrete event simulation is a set of circumstances. It is possible that two different event s occur simultaneously. The purpose of a *discrete event simulation* is to study a complex system by computing the times that would be associated with real events in a real-life situation.
Two basic methods exits for updating clock time. The first method is called "Interval-oriented" and another one is called "Event-oriented". Simulation has many types. But most important are Discrete Simulation and Continuous Simulation. Basically the five key features found in the software simulation model. There are also three approaches to describing the discrete simulation. Well-known examples of Simulation are Flight Simulators, Fleet Management and Business games. However, there are a large number of potential areas for Discrete Event Simulation. The combination of all entities in the system-those being served, and those waiting for service-will be called a queue and system is called queuing system. Mainly two type of queuing system we face single server, and multi-server queuing system.

## 4.9 Key words

Discrete system simulation, Event-oriented, Interval-oriented, Queuing Systems, Single-Server Queue, Queue Parameters, Item population, Queue size, Dispatching discipline The Multi-server Queue, Basic Queuing Relationships, *Little's Law,Time Sharing System.*

## 4.10 Self-Assessment Questions

Q.1 What is meant by the "System State" in a simulation.

**Hint:** The system state is the information needed to fully describe the system at any point in time. It is the set of values of all state variables in the system, the state variables being the attributes of all the entities (or objects of interest) in the system.

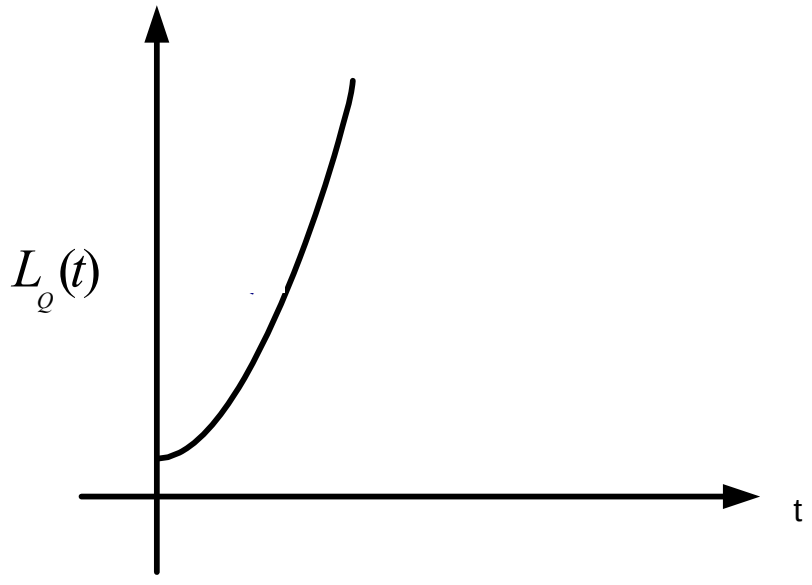
Q.2 The customer arrival process at a particular store is exponentially distributed with a mean arrival rate of 3 customers per hour. No customers have arrived in the last 5 hours. How many customers are expected to arrive in the next hour?

**Hint :**3, the expected number of future arrivals is independent of the past arrivals and, in particular, the time since the last arrival.

Q.3 A certain college has 5 printers in the computer room for the use of the college's 1500 undergraduate and 2000 graduate students. Print jobs arrive uniformly during the day and have an exponentially distributed page count. They are spooled on a single hard disk while waiting for the first printer to be available. In the standard notation of queuing systems, how would this queuing system be characterized? (e.g., as an M/M/1 queue? Or some other type?)

**Hint:** G/M/5/N/3500: General arrival, exponential service (assuming service time ~ length), 5 servers, a limit of N jobs in the queue (it has to be finite, based on the hard disk) and 3500 students in the population (again, finite). The hard disk is not a server, the printers are the servers, so c = 5, not 1.

Q.4 You observe the behavior graphed below in a queuing system. What observation can you make about *r*?




**Hint:** The queue length is growing, apparently without bound. The server utilization, *r*, is probably greater than 1.

**Q.5** Assuming an exponentially distributed service time, which queuing discipline is likely to have the shortest average waiting time: FIFO, LIFO, Priority, or SPT?

**Hint:** SPT (Shortest Processing First) will most often have the shortest average waiting time, since the server can process a large number of short jobs first. The longer jobs move to the rear of the queue and, although they will have longer waiting times, there are far fewer of them for exponentially distributed service times.

**Q.6** An arrival event occurs in a queuing system when the queue is not empty. What observation can you make about the status of the server?

**Hint:** It must be busy. If it were idle, it would have taken a waiting customer from the queue.

**Q.7** What are two events that can cause a change in a queuing system's state?
**Hint:** Customer arrival, service completion (customer departure)

**Q.8** We are simulating a packet switching system, where a linked list is being used to simulate the input buffer that stores one character per list record. The list contents at one point in time are shown in the table below. What is the data stored in the buffer if the "head" pointer value is 2?

| Array Position | Record | Next |
|---|---|---|
| 1 | space | 5 |
| 2 | C | 7 |
| 3 | C | 11 |
| 4 | E | 3 |
| 5 | I | 6 |
| 6 | N | 2 |
| 7 | O | 9 |
| 8 | R | 4 |
| 9 | R | 8 |
| 10 | S | 4 |
| 11 | T | 11 |

**Q.9** You are using a simulation to study the steady-state average waiting time of customers in a complex system. However, you find that, because of the simulation complexity, the computer time needed to arrive at a steady-state value is excessive, leading to poor estimates of the steady-state values. How might you improve the speed at which the simulation converges?

**Hint** Use an observation of real system to predict the average queue length. Use this as a starting position for the simulation. Alternatively, you could use prior steady-state values

as initial conditions for future simulations. As a third alternative, you could create a

simpler, faster simulation to estimate the steady-state values and use these as starting conditions.

**Q.10** The Third National Bank of Podunk has 6 tellers with an exponentially distributed service rate with a mean of 5 minutes. Customers arrive at the bank with exponentially distributed inter arrival times with an average inter arrival time of 1 minute. There is a single queue for customers to wait in, but the queue is restricted to 20 customers. What observation can you make about the departure rate of customers from the bank?

**Hint** The system has a limited queue length. Utilization of the tellers is high, but less than 1 – the average utilization is $\lambda/(c*\mu)=5/6$. The effective arrival rate $\lambda_e$ is reduced by customers who are turned away, so the departure rate must be reduced as well. Extra credit for calculating the actual departure rate. This would be $\lambda*(1-P_N)$ where N=20.

$$P_N = \frac{a^N}{c!.c^{N-c}}.P_0 \text{ ,} P_N = 7.929\times10^{-3}$$

$\lambda_\rho = \lambda.(1 - P_N)$ , $\lambda_\rho$

Q.11 For the Third National Bank example in problem 10, what is the probability that a customer shows up at the bank and is immediately served?

**Hint :**This is an M/M/c/N queue. We can calculate $P_0$ to find the chance of no wait time
$C = 0$ , $\lambda = 1$ , $\mu = 1/5$ , $a = \lambda / \mu$, $N = 20$
$\rho = \lambda / c.\mu$ , $\rho = 0.083$

$$P_0 = \left(1 + \sum_{n=1}^{c} \frac{a^n}{n!} + \frac{a^c}{c!} \sum_{n=c+1}^{N} \rho^{n-c}\right)^{-1} \qquad P_0 = 4.691\times10^{-3}$$

**Q.12** Customers calls arrive at a software support center with exponentially distributed inter arrival times with an average time between arrivals of 15 minutes. What is the probability that 5 customers will arrive between 1 and 2:30 p.m.?

**Hint:** Customer arrival process is exponential with arrival rate $\lambda = 4$/hour. This means that the number of customers arriving in a given time period is Poisson distributed.

$$P[N(t) - N(s) = n] = \frac{e^{-\lambda(t-s)}[\lambda(t-s)]^n}{n!}$$

the probability of 5 arrivals in 1 1/2 hous is : $\frac{e^{-6}[6]^5}{5!} = \frac{.0025 * 7776}{120} = .1606$

## 4.11 References/ Suggested Readings

1. Emshoff, J.R.and R.L.Sisson, Desing and Use of Computer Simulation Models, Macmillan Co. New York, 1970.
2. *G.Gordon. "System Simulation",PHI.*
3. *N.Ddeo, 'System Simulation with Digital Computer",PHI.*
4. Ball, P.: 'Introduction to Discrete Event Simulation', University of Strathclyde.
5. Pidd, M.: 'Computer Simulation in Management Science', John Wiley & Sons, Inc.
6. Robinson, S.: 'Successful Simulation: A Practical Approach to Simulation Projects', McGraw-Hill International (UK) Ltd.
7. Mendel Rosenblum , Edouard Bugnion , Scott Devine , and Stephen A Herrod ," Using the SimOS Machine Simulator to Study Complex Computer Systems", Computer Systems Laboratory, Stanford University.

**Subject : System Simulation and Modeling** **Author : Jagat Kumar**
**Paper Code: MCA 504** **Vetter : Dr. Pradeep Bhatia**
**Lesson : Continuous Simulation**
**Lesson No. : 05**

**Structure**



## 5.0 Objective

Up to last few units we have studied that simulation is of two kinds, discrete and continuous. In last unit we already know about discrete event simulation, in this unit we will study about continuous simulation.

## 5.1 Introduction

So far, we saw discrete event simulations. This is useful when our problem is like a queue of events, sorted by the simulation time at which they should occur. Things are not always that simple. A continuous system is one in which the predominant activities of the system cause smooth changes in the attributes of the system entities. When such a system is modeled mathematically, the variables of the model representing the attributes are controlled by continuous functions. More generally in continuous system the relationships described the rates at which attributes change so that the model consists of differential equations.

Continuous simulation is something that can only really be accomplished with an analog computer. Using a digital computer one can approximate a continuous simulation by making the time step of the simulation sufficiently small so there are no transitions within the system between time steps. The premise for a continuous simulation is that there is a continuous time flow and the simulation is stepped in time increments.

Differential equations, both linear, nonlinear, ordinary and partial occur repeatedly in scientific and engineering studies. The reasons for this prominence is that most physical and chemical processes involve rates of change, which require differential equations for their mathematical description.

## *5.2 Continuous Simulation*

**Formal Definition:** Continuous simulation concerns the modeling over time of a system by a representation in which state variables change continuously with respect to time.

Typically, we use differential equations (already discussed in Unit III) that give relationships for the rates of change of the state variables with time. So, how do we solve these systems of deferential equations? in very easy cases these can be solved analytically otherwise solved numerically.
The simplest differential equation models have one or more linear differential equations with constant coefficients. It is then often possible to solve the model without the use of simulation. Even so, the labor involved may be so great that it is preferable to use simulation techniques. However, when nonlinearities are introduced into the model, it frequently becomes impossible or, at least, very difficult to model these systems. The methods of applying simulation to models where the differential equations are linear and have constant coefficients, and then generalizing to more complex equations.

### *5.2.1 Examples of Continuous Simulation*

**Example** 1 : Consider an easy predator-prey model. Let the prey population at time t be given by x(t), and the predator population by y(t). Assume that, in the absence of predators, the prey will grow exponentially according to $x' = ax$ for a certain a > 0. We also assume that the death rate of the prey due to interaction is proportional to x(t)y(t), with a positive proportionality constant. So:

$$x'(t) = ax(t) - bx(t)y(t)$$

Without prey, predators will die exponentially according to $y' = -cy$ for a certain c > 0. Their birth strongly depends on both population sizes, so we finally find for a certain d > 0:

$$y'(t) = -cy(t) + dx(t)y(t)$$

$$\left\{\begin{array}{l} x'(t) = ax(t) - bx(t)y(t) \\ y'(t) = -cy(t) + dx(t)y(t) \end{array}\right\}$$

We immediately see that both ($e^{at}$, 0) and ($0,e^{-ct}$) are solution of (x(t),y(t)). From this system we find that for every solution we must have

$$x'\left(\frac{c}{x} - d\right) + y'\left(\frac{a}{y} - b\right) = 0$$

Integrating both sides of above equation gives us
c log x(t) –dx(t) + a log y(t) – by(t) = constant
Solutions for a = b = c = d = 1

The given model is considered very simple. Why? the integrating we did two above is possible, rest of the world has no influence, no randomness involved. Usually we cannot find closed-form solutions for the system of differential equations.
How do we deal with this problem? Solve numerically.

Example 2

Suppose we consider the example of the interaction of the principle and interest associated with a savings account. This can be represented by a systems thinking Figure 1 as follows:

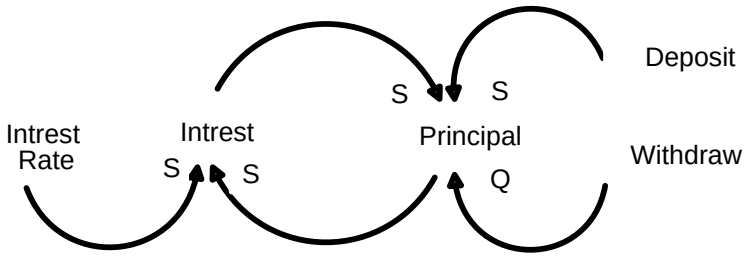



**Figure 1: Interaction of the Principle and Interest**

This diagram indicates that **Deposits** increase the **Principal** and **Withdraws** decrease the **Principal**. Also, the **Principal** interacts with the **Interest Rate** (Refer Figure 2) on some periodic basis to create **Interest**. The **Interest** then serves to increase the **Principal**.
If we then turn this into a 10 year simulation with the assumptions that the **Principal** is initially 100, there are no **Deposits** or **Withdraws**, and an **Interest Rate** of 5% is paid once a year it might look like this in Extend.

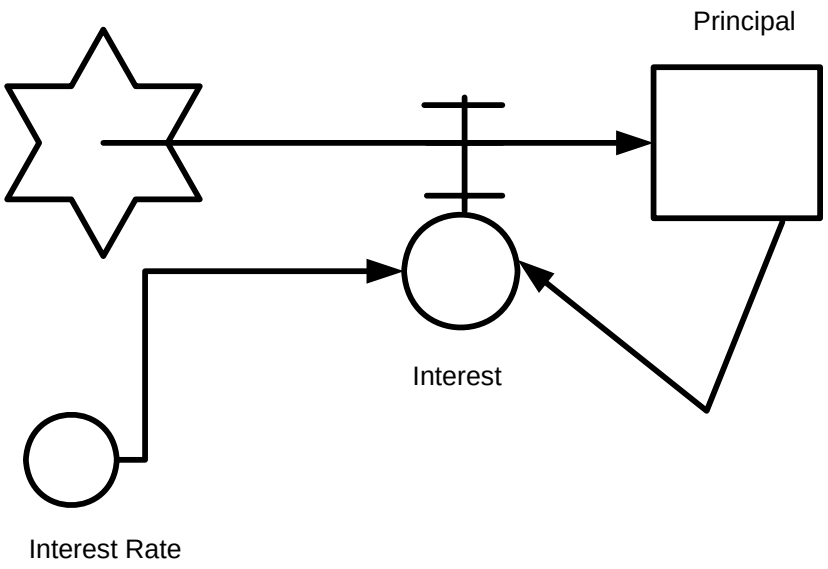



**Figure 2: Interaction between Interest rate and Principal**

With the following equations:

- Principal(t) = Principal(t - dt) + (interest) * dt
- Principal = 100
- interest = Principal * Interest_Rate
- Interest_Rate = .05

After a period of 10 years the Principal is about 155, and if we draw the graph between principal and time (year) that will looks continuous. This is because the simulation was stepped in increments of 1 year which is the same period over which the Interest is computed and applied to the Principal.
If we take the same simulation and run it in increments of 1 month, which means the Interest is calculated every 12 months the equation set becomes

- Principal (t) = Principal (t - dt) + (interest) * dt
- Principal = 100
    - interest = if ( (TIME/12)*12=TIME ) then Principal * Interest_Rate else 0
- Interest_Rate = .05

The indication is that the operation of this simulation appears continuous or discrete depending on the time frame over which we view the interaction even though each run is essentially a continuous simulation because of the equal time steps used.

## 5.2.2 Why do we do this (Continuous Simulation)?

We have discussed continuous systems whose process of evolution depends on differential equations. Such a system contains a number of parameters that must be estimated (for instance, the a, b, c, d > 0 in the

example of predator-prey model). Usually point estimates are calculated and used in the model. These estimates typically have uncertainty associated with them. We can incorporate uncertainty in our differential equations. Using fuzzy numbers as estimates of the unknown parameters, which will be discussed in unit VII.

## 5.3 The Uses of Simulation

There are differennt uses of simulation in almost all branches of science and engineering , social sciences , education etc., some of uses are decreibed in following section.

### 1. *Simulation in Science and Engineering Research*

Simulation has changed in a very fundamental sense, the way in which research is conducted today. Earlier most experiment carried out physically in the labs. Thousands and even millions were spending on physical model. Today a majority of these experiments are simulated on a computer. 'Computer experiments' besides being much faster, cheaper, easier, frequently provide better insight into the system then labs experiments do. Not all labs experiments, of course, can be replaced with the computer simulation. But 80-90 per sent of the experiment has been done on the computer. Best example of nuclear bomb simulation in a lab environment, which do not require any physical experiment of nuclear bomb in reality. For such kind of simulation we have to collect data from real experiment and then model the bomb and perform simulation. India having such capability to perform such kind of lab simulation of a nuclear bomb .

### 2. *Simulation in Soft Science*

Simulation can be expected to play even a more vital role in biology, sociology, economics, medicine, psychology, etc., where experimenting could be vary expensive, dangerous, or even impossible. In these area, the mathematical theories are even has less developed then in physical sciences. Moreover, in the field such as biology and economics the problem truly large, involving ten of thousand of variables. The complication causes by uncertainty are also greater in these areas than in physical sciences. The simulation has become an indispensable tool for a modern researcher in most social, biological and life sciences. Best example of simulation in medical sciences is drug design. Earlier after invention of a drug it took lot of time to experiments on monkeys, rats, and horses. After suitable experiments drugs launches in market. But now a days all the experiments performed using simulation and modelling techniques, which save, not only time but also save life of innocent animals, which frequently die during drug testing.

### 3. *Simulation in Business*

There are many problem faced by management that cannot be solved by standard operations research tool like linear and dynamic programming, inventory and queuing theory. Therefore, a business executive had to make decisions based solely on his intuitions and experience. Now he can use computer simulation to make better more meaningful decisions. Utilizing the power of a digital computer, he can build and study a simulation model containing high order complexities and a huge number of interdependencies, as well as uncertainty.
Some more uses in business process

- Impact of connection bank redesign on airport gate assignment
- Product development program planning

- Reconciliation of business and systems modeling
- Personnel forecasting and strategic workforce planning

**4.** ***Use of Simulation in Medical Education***

Simulation training has become an integral part of medical education at the best schools in America, which is why **West Virginia University** (WVU) plans to build its own comprehensive Clinical Simulation Center. Whether it's computer-based systems for improving intubations, simulated OR suites or virtual humans who mimic cardiac distress, WVU simulation labs will help medical, nursing, pharmacy and dentistry students improve their analytical, diagnostic and intervention skills. Center plans also include mobile units that will travel to rural settings, offering hundreds of healthcare practitioners the opportunity to learn skills that enhance the proactive of medicine.
In the same way flight simulators make real-life decisions easier for pilots and astronauts, mannequins, who look and act like real patients, enhance the real-world skills of all medical learners.
In India also, CBSE board banned the dissection of frog for biology students, instead of this fog dissection is performed in simulated environment, which save money, time and above all it save life of innocent fog.

**5. The Use of Simulation in Road Transport Incident Detection**

Automatic incident detection is becoming one of the core tools of urban traffic management, enabling more rapid identification and response to traffic incidents and congestion. Existing traffic detection infrastructure within urban areas (often installed for traffic signal optimization) provides urban traffic control systems with a near continuous stream of data on the state of traffic within the network. The creation of a simulation to replicate such a data stream therefore provides a facility for the development of accurate congestion detection and warning algorithms.

It describes firstly the augmentation of a commercial traffic model to provide an urban traffic control simulation platform and secondly the development of a new incident detection system (RAID—Remote Automatic Incident Detection), with the facility to use the simulation platform as an integral part of the design and calibration process.

***6.*****Simulation in Ground Water Management Process**

No body can know the status of ground water unless other wise we dig the earth deeply . The problem is more sever in drought-affected areas especially in India but now we can studied the status of ground water by modelling and simulation techniques.
Several computer codes have been developed during the past two decades to facilitate ground-water simulation-optimization modeling these codes differ in the numerical model used to represent the ground-water flow system and the types of ground-water management problems that can be solved.
The new process, which is called the Ground-Water Management (GWM) process can be applied to a broad range of ground-water management problems, such as limiting ground-water-level declines or stream flow depletions, managing ground-water withdrawals, and conjunctively using ground water and surface water. Management variables that can be specified in GWM include withdrawal and injection wells, artificial-recharge basins, and imports and exports of water. The types of constraints that can be specified include upper

and lower bounds on pumping and injection rates, water-supply demands, hydraulic-head constraints such as draw downs and hydraulic gradients, and stream flow and stream flow-depletion constraints. GWM uses a widely applied technique called the response-matrix approach to solve ground-water management problems.
Simulation can also be used in estimation of the runoff from rain fall, calculation of soil moisture, infiltration and the movement of moisture in overland flow of stream, crop modelling and weather forecast.

### *7.* The Use of Simulation in Manufacturing System Design and Operation

Manufacturing system design involves making long-term decisions. As such, it is practical to spend more time analyzing alternatives than would be the case for operational decisions. It was in the general manufacturing system design area that simulation initially gained popularity. Initially, models used for these projects were typically coded using general-purpose programming languages. The successful application of simulation in this area led to the development of specialized subroutine libraries and, later, comprehensive simulation languages and software packages. These developments have significantly simplified the use of simulation and have led to its more general use. The general class of manufacturing system design has been further subdivided into facility design, material handling system design, manufacturing cell design, and flexible manufacturing system design.
Some more uses of simulation in manufacturing

- Analysis of electronics assembly operations
- Design and evaluation of a selective assembly static for high-precision scroll compressor bells
- Comparison of dispatching rules for semiconductor manufacturing using large-facility models
- Evaluation of cluster tool throughput for thin-film head production
- Determining optimal lot size for a semiconductor back-end factory
- Optimization of cycle timer and utilization in semiconductor test manufacturing
- Analysis of storage and retrieval strategies in a warehouse
- Investigation of dynamics in a service oriented supply chain
- Model for an Army chemical munitions disposal facility

### 8. Simulation Techniques in the Social Sciences

Computer simulation as an idea has a history that goes back to the beginning of computers, but if we look specifically at the history of the simulation of societies, the history is much shorter. During the late fifties and early sixties, there was quite a lot of research that tried to model macro-processes embracing the whole world. The best-known example is the Club of Rome work by Meadows et al (1972) using ideas developed by Forrester (1971). This constructed a model interrelating population, pollution levels, the availability of natural resources and capital stocks. There were also other efforts to do social simulation during this period. These all efforts are conducted to know the questions like, what would happen to the ecology of the world over the

next fifty years. To achieve such predictions, however, you have to make a great number of assumptions about how people would live and thus about the values of the many parameters in the model. The majority of these assumptions could not easily be justified. One of the major innovations in the late 1980s and 90s was the development of simulations based on multiage modeling, in particular bringing in ideas from artificial intelligence and cellular automata.

There are so many other areas where we can use the simulation techniques

1. ***Construction Engineering***
    - Construction of a dam embankment
    - Trenchless renewal of underground urban infrastructures
    - Activity scheduling in a dynamic, multi-project setting.
    - Investigation of the structural steel erection process
    - Special-purpose template for utility tunnel construction

2. ***Military Applications***
    - Modeling leadership effects and recruit type in an army recruiting station
    - Design and test of an intelligent controller for autonomous underwater vehicles
    - Modeling military requirements for non-war-fighting operations.
    - Multi-trajectory performance for varying scenario sizes
    - Simulation of a missile trajectory
    - Simulation of a fighter aircraft training

3. ***Logistics, Transportation, and Distribution***
    - Evaluating the potential benefits of a real-traffic planning algorithm
    - Evaluating strategies to improve railroad performance
    - Parametric modeling in rail-capacity planning
    - Analysis of passenger flows in an airport terminal
    - Proactive flight-schedule evaluation
    - Logistics issues in autonomous food production systems for extended-duration space exploration
    - Sizing industrial rail-car fleets
    - Product distribution in a newspaper industry
    - Design of a toll plaza
    - Choosing between rental-car locations

4. ***Human Systems***
    - Modeling human performance in complex systems
    - Studying the human element in air traffic control.

## 5.4 Summary

Simulation is generally more adequate because it involves fewer approximations than conventional methods. Simulation allows adjustment for change, which conventional methods cannot do effectively. In any case, if the time and cost are measured against the quality and completeness of the results, simulation is far ahead of the conventional techniques. Even though the available data are limited, simulation can still be useful because the data are used in a physically rational computational program.

## 5.5 Key Words

Manufacturing Systems Design,Artificial evolution,Virtually Reality ,Soft Science, Automatic Incident Detection,Hydrology, Continuous Simulation.

## 5.6 Self-Assessment Questions

Q.1 In a chemical reaction one molecule of a substance X is produced for one molecule each of substance A and B. The initial concentrations of A and B are a and b, respectively. Let x be the concentration of X and assume that it is initially zero. The are at which x increases is 0.1 times the product of the current concentration of A and B. Assume a and b initially 0.8 and 0.4, respectively, simulate the production of X.

Q.2 Identify six different problems from your own experience that you think should be solved using continuous simulation.

Q.3 Describe the use of simulation in education area.

Q.4 Describe the use of simulation in area of weather forecasting.

Q.5 Describe the use of simulation in Medical Science?.

Q.6 In a field there are four animals – a dog, a mongoose, a snake and a mouse. Dogs kill mongooses, mongooses kill snakes, and snakes kill mice. The speeds of the mouse snake mongoose are, respectively, 8,12,18 and 30 km/hr. Simulate the chase with the different starting position to see which animal gets kill first.

Q.7 Write the benefits of simulation in daily life.

## *5.7 References/ Suggested Readings*


8. Axel rod, R. (1997) 'Advancing the art of simulation in the social sciences' in R. Conte, R.
9. Ray K. Linsley and Norman H. Crawford 'Earliest hydrologic simulation models'
10. Emshoff, J.R.and R.L.Sisson, Desing and Use of Computer Simulation Models, Macmillan Co. New York, 1970.
11. *G.Gordon. 'System Simulation'*
12. *N. Deo, 'System Simulation with Digital Computer", PHI.*

**Subject : System Simulation and Modeling Author : Jagat Kumar**
**Paper Code: MCA 504 Vetter : Prof. Dharminder Kumar**
**Lesson : Simulation Language**
**Lesson No. : 06**

**Structure**

## 6.0 Objective

As we learn from previous units, computer simulation is useful technique, to study a wide variety of problems related to almost all fields, ranges from science and engineering to business and social science. The abstract model of the system converted into suitable algorithm and finally converted into an executable computer program in a suitable programming language, then after only the system behaviour is understood by analyzing the output results, as the program runs. General-purpose high-level language can be used for simulation but now a day specific simulation languages are also available in market. In this unit we will learn the overview of different simulation language with detailed study of SIMULA 67.

## 6.1 Introduction

As we learn in previous units about the continuous and discrete event simulation. These are two important type of simulation. The simulation language depending around these two types where one is continuous simulation languages and another one is discrete simulation language. Continuous simulation languages developed in late fifties as simulators of analogue computers. Continuous and discreet simulation language can be classified in several ways.

## 6.2 Continuous Simulation Language

Continuous simulation languages developed in late fifties as simulators of Analogue computers. Simulation on analogue computers is based on creating an analogue electronic system whose behavior is described by the same mathematical model (set of differential equations) as the system being investigated. The main problem of analogue computers is an analogue implementation of certain operations like multiplication, generation of some functions, generation of delays and others. Digital computers perform all these functions very easily and today continuous simulation is performed only on them. Nevertheless there is one operation where the analogue computers are better, which is integration. Digital computers use numerical integration that is generally slower and less accurate compared with the integration of an analogue integrator. Some special applications based on fast response use therefore the so-called hybrid computers that contain analogue and digital parts connected by Analog/Digital and Digital/Analog converters. The digital part does everything except integration. It computes inputs of integrators that converted by Digital/Analog converters to analogue signals inputted to analogue integrators. Their outputs are treated in the opposite way. The digital part also controls the interconnection of the analogue part that might thus change during computation.

### 6.2.1 Classification of Continuous Simulation Languages

Continuous simulation language can be classified in two ways, as

1. Block oriented simulation languages
2. Expression oriented continuous languages

**1. Block Oriented Simulation Languages**

Block oriented simulation languages are based on the methodology of analogue computers. The system must be expressed as a block diagram that defines the interconnection of functional units and their quantitative parameters. "Programming" means entering the interconnection of the blocks and their description. Then the user adds statements and/or directives that control the simulation. If the system is described as a set of equations, they must be converted to a block diagram. This conversion is a simple straightforward process. The typical blocks available in most continuous block oriented languages are integrators, limiters, delays, multipliers, constant values, adders, holders, gain (coefficient) and other.

**2. Expression Oriented Continuous Languages**

Expression oriented continuous languages are based on writing expressions (equations) that represent the mathematical model. So the system simulated must be expressed by a set of equations. Then the user adds statements and/or directives that control the simulation. Some languages enable both block and expression based ways of system definition. Simulation control means selection of the integration method (because some languages offer more), the integration step, the variables (outputs of blocks) that should be observed, the intervals for collecting data for printing and/or plotting, scaling of outputs (that may be also done automatically), duration of the simulation runs, number of repetitions and the way certain values are changed in them.
Models are created using a simple continuous simulation environment based on Expression oriented approach and can be easily modified to model any other systems described by differential equations.

## 6.3 Discrete Simulation Language

Discrete simulation deals with systems whose dynamics can be considered (due to the level of abstraction) as a sequence of events at discrete time points. The key point of a discrete simulation language is the way it controls the proper sequencing of activities in the model. This is also the way a user must "view the world" when using the language and a base for classification of discrete simulation languages.

### 6.3.1 Classification of Discrete Simulation Languages

Discrete simulation language can be classified in four ways which described below

1. Flowchart Oriented Languages
2. Activity Oriented Languages
3. Event Oriented Languages
4. Process Oriented Languages

**1. Flowchart Oriented Languages**

Flowchart oriented languages are represented by the language GPSS (General Purpose Simulation System), that exists in many versions on various computers. The user must view the dynamics of the system as a flow of the so-called transactions through a block diagram. Transactions are generated, follow a path through a network of blocks, and are destroyed on exit. In blocks transactions may be delayed, processed, and passed to other blocks. Blocks are in the program represented by statements that perform the activities of the model.

**2. Activity Oriented Languages**

Activity oriented languages are not based on explicit scheduling of future activities. For each activity the user describes the condition under which the activity can take place (that also covers scheduling if the condition is reaching certain time). The algorithm of the simulation control repeatedly increments time and tests conditions of all activities. The disadvantage of this approach is obvious. It is necessary to evaluate all conditions in every step that may be very time consuming. On the other hand it is conceptually very simple and the algorithm can be easily implemented in general high-level languages (there are simulation languages based on this approach, but not widely used). The models of a simple queuing system that demonstrate the activity oriented approach. These models are accompanied by several units that implement operations on two way linked lists that are later used to implement stacks and queues.

**3. Event Oriented Languages**

Event oriented languages are based on direct scheduling and canceling of future event. The approach is very general. The user must view the dynamics of the system simulated as a sequence of relatively independent events. Every event may schedule and/or cancel another event. The system routine must keep record of scheduled events. That's why every event is represented by the so-called event notice that contains the time, the event type, and other user data. Event notices are kept in the so-called calendar, where the event notices are ordered by the scheduled time. After completion of an event routine, the system removes the event notice with the lowest time from the calendar, updates the model time by its time, and starts the corresponding routine. This is repeated until the calendar becomes empty or the program stops because of other reason. Scheduling means inserting event notices to the calendar by the scheduled time, canceling removes them. The approach based on explicit expressing of events is called Discrete Event Simulation that is sometimes generalized to discrete simulation as such. A typical representative of this group of languages is the language SIMSCRIPT.

**4. Process Oriented Languages**

Process oriented languages are based on the fact, that events are not independent. An event is typically a consequence of other previous events. In other words it is often possible to define sequences of events that may be viewed as entities of a simulation model at higher level of hierarchy. A sequence of events is called process. Unlike events process has a dimension in time. Process based abstract systems are very close to reality that is always made of various objects that exist and act in parallel interfering with each other. Process way of viewing system dynamics is thus very natural. Mostly a process models an activity of a real object. It is believed, that process oriented discrete simulation is the best way how to create discrete simulation models. Typical representatives of this group of languages are MODSIM, SIMSCRIPT II.5, and the system class SIMULATION of the Simula language.

## 6.4 Other Simulation Languages

There are other simulation languages exists

1. Object Oriented Simulation language
2. On Line Simulation language
3. Advanced Continuous Simulation Language
4. Graphic **Simulation** Language (GSL) - a combined continuous and discrete simulation language

**1. Object Oriented Simulation**

Object Oriented Simulation (OOS) can be considered as a special case of Object Oriented Programming (OOP). Some principles of OOP like existence of a varying number of instances of interfering objects have been in standard use in simulation environment for a long time, often using other terminology. The Simula language (used to be called Simula 67) is the first true object oriented language. OOPS like classes, inheritance, virtual methods, etc. have been defined in Simula. MODSIM is another object oriented simulation language.

These are the most commonly accepted features of OOS

a. The algorithm or system dynamics is expressed in terms of objects (actors) that exist in parallel and that interact with each other. Every object is represented by:

1. Parameters
2. Attributes
3. Methods
4. Life, that represents the activity started upon object creation.

Objects can interact in these ways

1. Direct access to parameters and attributes
2. Mutual calling of methods
3. Communication and synchronization of objects lives.

b. Conceptually a object is defined as Object = **Data** + **Procedures** that is called Encapsulation. Generally the object's data or a part of it, is hidden and values can be accessed and modified only through (well defined) methods. This concept is called Information hiding.

c. Similar objects (actors) are grouped by to the classes also called prototypes. A class describes objects that have the same parameters, attributes, methods, and lives. A class can be also interpreted as knowledge of certain type of objects. Such knowledge is represented by a data part and by operations that can be performed on the data. This is similar to abstract data types, but classes are much richer.

d. Objects can be classified hierarchically generally called inheritance. Very often the term subclass is introduced. A subclass Y of a class X inherits all parameters, attributes, and methods from the class X. Its declaration can add any number of additional parameters, attributes, and methods. A subclass may also add some activity to the life of the parent class. A subclass can be used as a parent class of other subclasses. Some OOP languages (not Simula) enable the so-called multiple inheritance. In this case a subclass can inherit from more than one parent classes. It might be desirable, that certain methods then behave in different way according to the current object instance being referenced that may change dynamically during program execution. This concept called polymorphism is supported by the mechanism called late binding and the methods involved are called virtual methods that may change at every level of hierarchy.

**2. On Line Simulation**

Internet together with Java and JavaScript offer incredible possibilities in problem solving. Instead of time consuming downloading and installation of software packages, it is possible to open directly various solvers, especially for problems that are not frequent and that do not require time consuming computation.

**3. Advanced Continuous Simulation Language**

The Advanced Continuous Simulation Language, or ACSL (pronounced "axle"), is a computer language designed for modelling and evaluating the performance of continuous systems described by time-dependent, nonlinear differential equations. It is a dialect of the Continuous System Simulation Language (CSSL).
ACSL is an equation-oriented language consisting of a set of arithmetic operators, standard functions, a set of special ACSL statements, and a MACRO capability which allows extension of the special ACSL statements. ACSL is intended to provide a simple

method of representing mathematical models on a digital computer. Working from an equation description of the problem or a block diagram, the user writes ACSL statements to describe the system under investigation.The important feature of ACSL is its sorting of the continuous model equations, in contrast to general purpose programming languages such as FORTRAN where program execution depends critically on statement order. Applications of ACSL in new areas are being developed constantly. Typical areas in which ACSL is currently applied include control system design, aerospace simulation, chemical process dynamics, power plant dynamics, plant and animal growth, toxicology models, vehicle handling, microprocessor controllers, and robotics.

**4. Graphic Simulation Language GSL**

GSL is a FORTRAN-oriented language, which combines the activity and process concepts of a discrete simulation language with continuous simulation concepts, thereby permitting the simulation of systems, which call for combining continuous and discrete simulation techniques. The basic structural component of GSL is the simulation block, which corresponds either to an activity of a discrete system or a dynamic region of a continuous system. Both discrete and continuous simulation blocks may have multiple process instances, which may be controlled dynamically at run- time. The result is a combined language, which retains the features of both continuous and discrete simulation languages and moreover takes advantage of the desirable features of each to supplement the other.

## 6.5 Introduction of SIMULA

The first Object Oriented Language (OOL) Simula 67 was officially introduced by Ole Johan Dahl and Kristen Nygaard at the IFIP TC 2 Working Conference on Simulation Languages in Lysebu near Oslo in May 1967. All modern programming work carried out today is based on principles of OOP introduced for the first time in the Simula definition.

A computer program is a series of instructions, which contains all the information necessary for a computer to perform some task. It is similar to a knitting pattern, a recipe or a musical score. Like all of these it uses special shorthand, known in this case as a programming language. We will learn the programming in SIMULA 67,in SIMULA 67, 67 stood for 1967, the year in which this earlier version was first defined .We will call SIMULA 67 as SIMULA only in our discussion.

### 6.5.1 Basic Building Block of SIMULA

SIMULA program is made up of sequences of instructions known as blocks, which act independently to varying degrees, but which are combined to produce the desired overall effect. The simplest programs contain only one **block,** known as the program block. All other blocks follow the same basic rules, so let us have a look at the single block program, example 6.1.

**Example 6.1: A simple example**

```
begin
```

```
  integer Int1;
  comment The first SIMULA program written for this book;
  Int1:=3;
  OutInt(Int1,4);
  OutImage
end
```

This is not the simplest possible program. We could have written

```
begin  end
```

and had a complete program block. This tells us the first rule about all blocks. **A block always begins with the word *begin* and ends with the word *end*.** *Begin* and *end* are called **"keywords",** since they are reserved for special purposes and may not be used for anything else.

SIMULA is not a case sensitive language which mean we can use lower case or upper case or mix of two to write program. **Instructions** in the SIMULA are of two types, **DECLARATIONS** and **STATEMENTS**. Every executable statements in SIMULA end with a semicolon. There are also pieces of the text which are ignored by the SIMULA system and which simply help any human who reads the program to follow what is happening. **These are called comments.**

SIMULA c**omments** begins with the **keyword *comment***, is followed by some text and ends in a semi-colon.

**comment The first SIMULA program written for this book;**

comments can also be used in certain places within instructions or combined on the same line as them. To help everyone understand your programs, you should include comments in them.

### 6.5.2 Declarations

In example 6.1, the first instruction after the word begin is a declaration:

```
integer Int1;
```

Declarations in a block show how much of the computer's memory is going to be needed by that block. They also show into how many sections it is to be divided , how big each is to be and to what sort of use each will be put. Each of these sections is given a name, by which the program will refer to it.

The program block in example 6.1 has only one declaration. It starts with the "type" **keyword *integer***, followed by a name or "identifier", Int1.

In example 6.1 it will be used to hold a whole number, which is called an integer in SIMULA.

The **identifier** Int1 is now the name of the space reserved in this way. We may sometimes refer to the value held in this space as Int1 also.

If we want to use more locations, we can declare them in the same way, being careful to give the correct type to each. Thus we might write

```
begin
  integer Int1;
  real Real1;
```

etc.

Which gives us a location of type real and called Real1, which we can use in this block.

### 6.5.3 Statements

The other instructions in our example are all statements. They tell the SIMULA system what it is to do.The first one is an "assignment" statement. It tells the system to place something in one of the locations declared for use by this block. In this case the value three is to be stored in the integer location declared as Int1. Since this value is of a type which matches that declared for the location the statement is legal SIMULA.
The next statement uses something called OutInt, which is a **"procedure**" and is available in all SIMULA systems. OutInt will write out the first number in the parentheses - ordinary brackets - after it. It writes it as a whole number at the end of the line, or "image", which is currently being created. The second number is used to tell the system how much space to use on the line, when writing out the first number.
The final statement uses OutImage. Like OutInt, OutImage is a procedure and is available in all SIMULA systems. Such standard procedures are known as **"system procedures".** They are not keywords, since you may declare a different meaning for them within a block if you wish.

### 6.5.4 Syntax Rules

The commonest errors reported by a compiler are those, which do not obey the grammar or "syntax" of the language. Often they are the result of typing errors.
The rule that a program block must start with *begin* and finish with *end* is a syntax rule.

**Syntax of Declarations**

The declarations

```
    integer Int1;
```

and

```
    integer Int1,Count;
```

both follow the syntax rules for declarations. A declaration has to be a keyword giving the type, followed by an identifier list. An identifier list is either a single identifier or a series of identifiers, separated by commas, with the option of as many spaces as desired either side of the commas.
The syntax rules for SIMULA, like those for most programming languages, are very strict. You cannot omit the space which indicates the end of the keyword *integer*, without breaking the syntax rules for a declaration.

Syntax of Identifiers

We have used the word identifier as the technical term for the name given to something in our programs. So far we have not considered what an identifier must look like.

The identifiers which we have used so far are

Int1
Count
OutInt
OutImage
Real1

Notice that procedure names are identifiers and follow the same rules. Keywords have the same syntax as identifiers, but they are not available for the programmer to define or redefine.
**An identifier is a sequence of letters, numbers and underline characters (the last are known sometimes as break characters). An identifier must start with a letter**.
Some systems set a limit on the number of characters in an identifier. Others allow long identifiers, but only look at the first so many characters. You should consult the documentation for any system before using it, especially when moving a program from one system to another.
Letters are often called "alphabetic characters", numbers "numeric characters" or "digits". Mixtures of these two types are called "alphanumeric characters".

The following are valid identifiers
TOTAL
A1
NEW4SUB6
MAX_SIZE
G43
I

Syntax of Blocks

A block starts with the keyword *begin*, which, like all keywords, must have at least one space following it or be the last word on a line.
This keyword is followed by a list of declarations or a list of statements or a list of declarations followed by a list of statements. Statements and declarations are separated by semi-colons or keywords. All declarations must come before any statements in a block.

The following are valid blocks

Example 6.2

```
begin
  integer I;
  real Arc;
```

```
    I := 5;
    Arc := 3.2;
    OutInt(I,3);
    OutImage
  end
```

In addition we can note that a block can be used in place of a simple statement. In this case it is called a **"sub-block"** or a block which is **"local"** to the program block.
A procedure call is an identifier (the name of the procedure) followed in some cases by a parameter list which is enclosed in brackets. The parameter list is a list of identifiers, constants and expressions, separated by commas.

**Examples 6.3: Procedure calls.**

**OutImage :** No parameters. Moves output to next line.
**OutImage :** One parameter, a text, which is printed on the current line. In this example a text constant is used as the parameter.
**OutInt(643,3):** Two parameters, both integers, separated by a comma. Prints out the first integer on the current line, padding on the left with spaces if this produces fewer figures than the second integer. Either or both the integer constants given as parameters in the example could have been replaced with identifiers or expressions.

Assignment statements have an identifier followed by optional spaces, followed by the assignment "operator", followed by optional spaces, followed by an expression. The assignment operator is the sequence colon followed by equal-sign, :=. Before giving an informal description of expressions, it is probably best to consider the examples of assignment statements given as 2.5.

**Examples 6.4: Assignment statements.**

```
Res := 3

Count := Last

Count := Last + 1

Message := "PLEASE TYPE YOUR NAME"

Next   :=   Average*Number + 2
```

The use of spaces before and after the assignment operator has no effect on the correctness of the statement. This applies to all "operators" in SIMULA.

**6.5.5 Expressions**

Several kinds of expressions are shown to the right of the assignment operator in these examples. The simplest of these is a constant, such as 3 or "PLEASE TYPE YOUR NAME". An expression can also be a single identifier, such as

```
Count := Last
```

The remaining examples show identifiers and constants separated by operators. Thus

```
Last + 1
```

is the identifier Last followed by the addition operator followed by the constant 1.

```
Average*Number + 2
```

is the identifier Average followed by the multiplication operator followed by the identifier Number followed by the addition operator followed by the constant 2. We shall not attempt a complete definition of expressions, but explain them as we need to use them. These examples should give a feel of what is meant.

A note on operators

The commonest arithmetic operators are given below.

```
+   Addition
-   Subtraction
*   Multiplication
/   Division with non-whole numbers (reals)
//  Division with whole numbers (integers)
:=  Assignment
```

**Semantics of Declarations**

**The same identifier cannot be declared twice in a block.**

The identifier is used to name a space in the computer's memory which is large enough to hold a value of the type specified. Whenever the identifier is used subsequently, it refers to this space and this type of value.

Semantics of procedure calls

**A procedure call must have the correct number of parameters.**

Each parameter must be of the correct type. The actions of the procedure are performed, using the parameters to provide information for the procedure if necessary.

Semantics of Assignments

**The type of the expression to the right of the assignment operator must be "compatible" with the type of the identifier on the left.**

The value of the expression on the right will be stored in the location reserved for the identifier on the left.

Semantics of Expressions

**The types of the quantities in an expression must be compatible. The type associated with the value of an expression is determined by the types of the quantities in it, except when the division operators are used. In this case the type is the same as the operator, i.e. integer for "//" and real for "/".**

Semantics of Blocks

Any identifier used in a statement in a block must already have been declared.
The statements in the block are performed in the order indicated.

Syntax and Semantics of a Comment

**A comment is a special sequence in SIMULA which is treated as a space by the compiler. It contains a message which explains what is going on around it in human terms. Its syntax is quite simple.**
A comment is:
The keyword *comment*, followed by any sequence of characters ending at the first semi-colon, ;.
or
the character !, followed by any sequence of characters ending at the first semicolon.
or
Any sequence of characters following the keyword *end*, ending at the first occurence of the keyword *end*, *else*, *when* or *otherwise* or a semi-colon or the end of the program.

Comments, partly to help you understand things and partly to get you into the habit of using them. They will help you see what is going on, but only in the important places.
Be very careful to end your comments properly. If you forget a semi-colon or keyword, as appropriate, you will lose some of the following program, since it will be treated as part of the comment. In particular, remember that the end of a line does not end a comment . Comments have no meaning in the program.

**Example 6.5 : The use of comments.**

```
begin
  comment Double space removal program,
        first version, Rob Pooley, March 1984;

      text T; ! Holds the text to be processed;
```

```
    InImage;                ! Reads the text into SysIn.Image;
    inspect SysIn do        ! Refer to SysIn not SysOut;
    begin
      T :- Blanks(Image.Length); ! See the next chapter;
      T:=Image;             ! Copies the characters into T;
    end;
    if T.GetChar=' ' then   ! First character is a space?;
    begin
      if T.GetChar=' ' then ! Second character is also?;
        T:=T.Sub(2,T.Length-1); ! Remove first character;
    end;

    comment Now write out the text;
    OutText(T);
    OutImage
  end Double space remover
```

### 6.5.6 Type Cast Actors

So far we have seen three types: integer, real and text. The only one we have looked at in any detail is integer. In fact SIMULA contains several other types as keywords or combinations of keywords. In addition it is possible for you to create combinations of these simple types and give them names, by using the ***class*** **mechanism**.

Types in Assignments

A type is given in a declaration so that the SIMULA system knows how much space to allow for the declared quantity. The system checks whenever a new value is stored into this location that the value is of the same type. If the types are different then the system may do one of two things.
It may "convert" the quantity being stored to that of the location. This is only possible between the types integer and real and their short and long variants.
If conversion between the types is not specified in SIMULA, the system will report a semantic error.
SIMULA will only convert one type to another where this has a clear meaning. In practice this is only the case for arithmetic values, i.e. types which represent numbers
Clearly we must be very careful how we use types. Even where we are allowed to mix types, we must be careful that we understand what will happen when a value of one type is converted into a value of another type.

Types of Parameters

Exactly the same rules apply where values are passed as parameters as apply when they are assigned. In effect the parameter given is assigned to a location with the type specified for that parameter.

Standard Types

We shall now look at each of the simple types provided in SIMULA.

**integer**

As we have seen, values are of type integer if they are whole numbers. They may be positive, negative or zero. On any particular SIMULA system there will be a largest positive and a smallest negative number, which can be held in an integer location. In our programs we have used integer constants to represent values being assigned to our integer locations. An integer constant is a whole number written as a sequence of decimal digits, i.e. any digit in the range 0-9. This may be preceded by a minus or, less commonly, a plus sign. A minus sign indicates a negative value; a plus sign has no effect.

Example 6.6: integer constants

```
2
45678231
-432
+ 1245
```

Spaces after the plus or minus are ignored. Spaces between digits are not allowed. It is also possible to give integer constants in other number bases.

**Real**

A real value is a number, which is not necessarily whole. It is held in a form, which allows it to contain a fractional part. In common speech it is what is known as a 'decimal' number or decimal fraction, i.e. it is written with a decimal point separating the whole and fractional parts of the number. A real value is restricted both by a largest/lowest range and by the number of significant decimal places, which can be held. Most of us are used to writing decimal numbers (decimal fractions) in what is technically known as **"fixed point" notation.** The examples of decimal constants used in examples so far are in this form. It can be described as two strings of decimal digits (in the range 0-9) separated by a full stop or period. Like integers they may be preceded by a minus or plus sign.

**Example 6.7: Legal fixed point real constants.**

```
5.7
236.0
3246.8096
-45.87
```

```
+ 46.876
```

The use of spaces is not allowed between digits or between a digit and the decimal point. Mathematicians often use another notation to write decimal values, especially where these are very large or very small. This way of writing them is known as "floating point" and is also allowed for writing real constants in SIMULA.

Character

A character holds a value, which represents a single character. SIMULA allows you to use any of the characters in the "character set" of the computer that you are using plus the characters defined by the **International Standards Organization (ISO)** character set standard. **(This is sometimes known as the ASCII character set.).** It is important to stress that a character has a different type from a text.
We normally think of a character as something written on a page. In the computing world this sort of character is often referred to as a "printing character". It is probably easy enough for us to accept that a space is also a printing character. It is rather harder to grasp that a character can produce other effects, such as making the printer start a new line or a new page. These are non-printing "control characters". Some of these character values produce very complex effects in certain printers or terminals and no effects or different effects in others. In general terms the values held in character locations are those, which are sent as instructions to printers, terminals and other hardware devices. Most of these instructions control the printing and formatting of written information. The simplest merely instruct the device to print a visible character. Character constants are normally written as single characters, enclosed in single quotes. Note carefully that a single character enclosed in double quotes is a text constant and may not be used as a character.
Control characters cannot be written in this way. They use their internal integer value written as an integer constant and enclosed in exclamation marks, inside single quotes. This notation can also be used to write printing characters, but is very clumsy.

Example 6.8: Legal character constants

```
'A'
'b'
'4'
'%'
' '
'"'
'!3!'          Control character, enclosed in exclamation marks.
```

Note that case, i.e. the difference between capital and small letters, is significant in character constants. Thus 'A' and 'a' are not equivalent. Note also how a single quote is represented as a character constant.

Boolean

A Boolean quantity can only have two values. These are True and False.
A Boolean can be assigned the value of any conditional expression and can be used wherever a conditional expression might be used, e.g. in an ***if*** **statement** or a ***while*** **loop**.
Boolean constants can only be represented by the keywords True and False.

**Text**

A text variable is used to refer to and manipulate sequences of characters. A sequence of characters is often known as a "string". In the examples using text variables so far we have often assigned strings as the values to be placed in locations declared as type text. From this it might seem that a text and a string are the same thing.
Text constants are strings, i.e. sequences of characters, surrounded by double quotes. Each character in a string can be any of those in the ASCII (ISO) standard set.
A character in a string can also be represented by its internal integer value enclosed in exclamation marks.
When a string is too long to fit onto a single line of your SIMULA program it can be continued on the following line by typing a double quote at the end of the first line and again at the start of the second line. These quote characters are ignored and the value of the constant is the string on the first line followed by the string on the second line.

**Example 6.9: Legal text constants.**

```
"The cat sat on the mat"

"34.56"

"%"

"This string is typed on two lines, but will be treated as if it "
"was on a single line, without the second and third double quotes."

"This string has a control character !10! embedded"

""""
```

Note that the last string shown in 6.9 will contain only one double quote. When you want to have a text constant, which contains one double quote, you must type two, so that the compiler knows that it is not the end of the string. Another example showing this is:

"This string contains only one "", but we must type two."

Note also that the single character text constants "%" and "" "" are not the same as the character constants '%' and ''. They have different types.

Initial values

An identifier of a certain type, which is not declared as a constant is often referred to as a "variable" of that type, since, unlike a constant of the same type, its value can be changed by assigning to it. When we declare such variable, the SIMULA system places an initial value in the location identified. This value is the same on all SIMULA systems on all computers for all variables of a given type.
Thus it is quite legal, and meaningful, in SIMULA to write

```
begin
  integer IntVal;
  OutInt(IntVal,4);
  OutImage
End
```

since the initial value placed in the location identified by IntVal will be printed. This will be the value initially given to all integer locations.
The values placed in each type of location are given below.

| | |
|---|---|
| integer | 0 |
| short integer | 0 |
| real | 0.0 |
| long real | 0.0 |
| character | The ISO NULL character, which is 'invisible'. |
| Boolean | False |
| text | NoText, the empty text, referring to an empty string. Equivalent to "". |

Implicit Conversion of Real and Integer

When a real value is assigned to an integer location or vice versa we have said that the value will be converted to one with the type of the location. When integers are converted to real, no problems are involved in understanding what will happen. The values 3 and 3.0 are usually thought of as identical and such a conversion presents no ambiguities.
On the other hand, when real are converted to integers, the result depends on how we deal with the fractional part of the real value. How do we get rid of the figures to the right of the decimal point?
If we go back to our earlier examples we might use them to produce example 6.9. Running this would show the outcome of implicit conversion of real to integer.

**Example 6.9: Implicit conversion and rounding.**

```
begin
  integer I1, I2, I3;
  I1 := 3.2;
```

```
    I2 := 3.9;
    I3 := 3.5;
    OutInt(I1,4);
    OutInt(I2,4);
    OutInt(I3,4);
    OutImage
  end
```

Clearly more than one outcome is possible.
If all reals are rounded up the output will be

```
   4   4   4
```

if down

```
   3   3   3
```

and if to the nearest integer value

```
   3   4   4
```

or

```
   3   4   3
```

the last two depending on whether 3.5 is regarded as nearer to 3

Constant Declarations

Constant declarations were introduced into SIMULA very late on and are still regarded as controversial by older SIMULA programmers. They should be used with some restraint if you want to move your programs to other systems. Older SIMULA systems will not have constant declarations.
A constant declaration allows an identifier to be assigned a value in its declaration. This identifier may not then be assigned another value. This can be very useful when using the same value frequently in a program, especially when the value is easily mistyped or has no obvious meaning.

```
real Pi = 3.14159;          ! A constant declaration;
```

### 6.5.7 Conditional Statements

The ability that makes computers more than calculators or fancy typewriters is the ability to perform different actions according to some condition or conditions, which it can determine to be either true or false. Computers can be told, "Check this condition. If it is true then do the following, otherwise do this other thing". In some cases the other thing is nothing. Put crudely, computers can make choices. What they cannot do, or, at least, as far as the author is aware, not yet, is decide which condition to test, or whether the actions which follow are really sensible. They must be told these things and programming languages have mechanisms for doing so. In SIMULA, the most important construction for making choices is the "conditional statement" or, as it is often known, the *if* statement.

**Example 6.10: Simple use of an *if* statement.**

```
begin
  integer Int1;
  Int1:=InInt;
  if Int1=2 then OutText("YES");
  OutImage
end
```

This program will read in a whole number and compare its value against 2. If it is equal to 2 then the program will print YES, otherwise a blank line will be printed. Compile and run it to make sure. From this example we can see the syntax of an *if* statement. An *if* statement starts with the keyword *if*, followed by a condition, followed by the keyword *then*, followed by a statement. The program checks the condition. If it is true, the statement is executed (or carried out, if you prefer), but if it is false, the statement is skipped. The next statement, if there is one, is then executed. An *if* statement may be used wherever a simple statement may be used.

**The *if-then-else* statement**

Consider example 6.11 which uses *if* statements.

**Example 6.11: Un-combined *if* statements.**

```
begin
  integer Int1;
  Int1 := InInt;
  if Int1=2 then OutText("YES");
  if Int1 ne 2 then OutText("NO");
  OutImage
end
```

Here we have added a second *if* statement to our first example, but all that it does is check the opposite condition to the first. ***ne*** is the symbol for "not equal" in SIMULA. This program will print out YES if the number read in is 2 otherwise it will print out NO.

### 6.5.8 Compound Statements and Blocks as Statements

As we mentioned in previous section that a block in SIMULA can be used as a statement. More generally, we can use a sequence of statements enclosed in *begin* and *end* wherever we can use a simple statement. In fact SIMULA uses the term "compound statement" for such a sequence if it contains only statements and the term block where it contains its own declarations. We are able to define our own procedures, which we can then use wherever we like inside the block where we declare them. In fact we can even build a library of our favorite procedures and use it in all our programs.

### 6.5.9.Declaring Procedures

It is clearly not enough to declare a procedure in the way we declare an integer. The declaration

```
procedure Concatenate
```

cannot magically tell the SIMULA system what we want Concatenate to do. We must also supply the actions to match the name. Here is a valid procedure declaration:

```
procedure PrintName;
OutText("Alice");
```

The syntax of the simplest procedure declaration is the keyword ***procedure***, followed by the identifier to be used for the procedure, followed by a semi-colon, followed by a statement. The statement following the semi-colon is known as the **procedure body** and specifies what is to be done each time the procedure is invoked or "called" in the subsequent program. Calling the procedure is done by using its identifier as a statement in the program, exactly as we call system procedures.

Example 6.12 shows the use of our procedure, PrintName.

**Example 6.12: Simple procedure use.**

```
begin

  procedure PrintName;
    OutText("Alice");

  Concatenate;
  OutImage
end
```

Note the use of blank lines to make it easier to see where the procedure begins and ends. These are not compulsory, but make the program more readable to humans. We would normally want to have more than one statement in our procedure body.

**Parameters to Procedures**

We have already seen how parameters can be used to pass values to system procedures. Example 6.13 shows how to declare two texts as parameters to the Concatenate procedure.

**Example 6.13: Concatenate with parameters.**

```
begin
  procedure Concatenate(T1,T2); text T1,T2;
  begin
    text T3;
    T3:-Blanks(T1.Length+T2.Length);
    T3:=T1;
```

```
    T3.SetPos(T1.Length+1);
    while T2.More do T3.PutChar(T2.GetChar);
    OutText(T3);
    OutImage
  end;
  text Text1,Text2;
  Text1:-"Fred";
  Text2:-"Smith";
  Concatenate(Text1,Text2)
end
```

This passes in our texts, T1 and T2, which are now used as parameters to the procedure. The identifier given for the procedure in its declaration is followed by a list of all the parameters to be used, enclosed in parentheses and separated by commas. The declaration text T1, T2;following the semi-colon and before the *begin* is known as the type specified and gives the type of each of the parameters. Where more than one type of parameter is to be used, more than one type declaration must be given. 6.14 is an example, using one text and one integer parameter.

**Example 6.14: A procedure with more than one type of parameter.**

```
begin
  procedure TextAndInt(T,I); text T; integer I;
  begin
    OutText(T);
    OutInt(I);
    OutImage
  end;
  TextAndInt("NUMBER",10)
end
```

Parameter Modes

The way of specifying parameters that we have used so far will always work for passing values into a procedure. If we want to get information out, we may have to add a mode specifier for some parameters. This sounds confusing, but is easy to follow in practice. Here is a final version of Concatenate.

**Example 6.15: Using a name parameter to return a result.**

```
begin
    procedure Concatenate(T1,T2,T3);
    name T3; text T1,T2,T3;
    begin
      T3:-Blanks(T1.Length + T2.Length);
```

```
    T3:=T1;
    T3.SetPos(T1.Length + 1);
    while T2.More do T3.PutChar(T2.GetChar);
  end;
  text Text1,Text2,Text3;
  Text1:-"Fred";
  Text2:-"Smith";
  Concatenate(Text1,Text2,Text3);
  OutText(Text3);
  OutImage
end
```

Notice that, as well as specifying that T3 is of type *text*, we have specified that it is *name*. *name* is not a type but a mode. When a parameter is defined as of name mode, any assignments to it alter the value of the variable actually passed in the call, rather than a local copy, as would have happened with the other parameters, which are passed by value. **In fact there are three modes; value, reference and name.** Where a mode specifier is not given for a parameter, a mode of value or reference is assumed, depending on its type. Some modes are illegal for certain types. Table 6.1 is a complete table of the assumed (usually referred to as default), legal and illegal modes for parameters to procedures.

| Type | Mode | Value | Reference |
|---|---|---|---|
| Simple type | Default | Illegal | Legal |
| text | Legal | Default | Legal |
| Object reference | Illegal | Default | Legal |
| Simple type array | Legal | Default | Legal |
| Reference type array | Illegal | Default | Legal |
| procedure | Illegal | Default | Legal |
| type procedure | Illegal | Default | Legal |
| label | Illegal | Default | Legal |
| switch | Illegal | Default | Legal |
| | Illegal | Default | Legal |

**Table 6.1: Complete table of the modes for parameters to procedures.**

A simple type is integer, real, character or Boolean and any long or short variants of them. Now we can see that name is always legal and reference is the default for all but simple types. Do not worry about the meaning of those types which are new. We shall consider their use when we encounter them.

Value Parameters

A value parameter to a procedure acts as if it were a variable of that type declared in the body of the procedure. The value passed to it when the procedure is called is copied into it as part of the call statement. Since values declared inside a block cannot be used

outside that block, the value of this mode of parameter is lost on returning from the procedure. When calling a procedure, any value of the correct type may be passed to a value mode parameter. Thus constants, expressions and variables are all allowed.

To see the effect of passing a parameter by value, consider example 6.16.

**Example 6.16: Passing parameters by value.**

```
begin
        procedure P(Val); integer Val;
        begin
          OutInt(Val);
          OutImage;
          Val := Val - 1;
          OutInt(Val);
          OutImage
        end..of..P;

        integer OuterVal;
        OuterVal := 4;
        P(OuterVal);
        OutInt(OuterVal);
        OutImage
      end
```

The value in OuterVal, 4, is copied into the parameter Val's location when P is called. Thus the first number printed will be 4. When 1 is subtracted from Val, OuterVal is not changed. Thus the second number printed is 3, but the third is 4. When a text is passed by value to a procedure (N.B. this is not the default) it has the effect of creating reference to a local text frame with the same length as the text passed, into which the characters from the latter text are copied. Consider example 6.17.
OutLine is actually quite a useful procedure. Note that in order to pass our text parameter by value we have to give a mode specification for it, using the keyword ***value*.** When the procedure is called, the parameter T is initialized as if the following statements had been executed.

```
T :- Blanks(OuterT.Length);
T := OuterT
```

**Example 6.17: Text parameter passed by value.**

```
begin
  text OuterT;

  procedure OutLine(T); value(T); text(T);
  begin
    OutText(T.Strip);
```

```
    OutImage
  end..of..OutLine;

  OuterT:-"Here's a line";
  OutLine(OuterT)
end
```

Reference Parameters

When a parameter is passed by reference, the local parameter points at the location holding the object passed, rather as if the reference assignment operator had been used. No local copy is made of the contents of the object. For every reference parameter type except text, this explanation is sufficient and should be reconsidered for its meaning when those types are encountered. As we have seen, when a text is assigned by reference new copies of Pos, Length etc. are made, but the same actual text frame is referenced. Pos, Length etc. will have the same values as those for the original reference, but will not change if the originals do. As far as the passing of text parameters by reference is concerned the following effects occur:

The characters in the frame referenced by the parameter may be changed by the procedure. Since this is the same actual location as the frame of the reference, which was copied, the contents of the frame remain changed when execution of the procedure is complete.
The other attributes have local versions created, with the same values as those current for the parameter. When those other attributes are changed for the parameter, they remain unchanged for the original. Thus, any changes to these are lost when execution of the procedure is complete.
Try rewriting the Concatenate procedure (Example 6.13) with all the parameters passed by reference. What would be the effect on running the program using it now? .You should find that it fails since the Length of Text3 cannot be changed by manipulating T3 inside the procedure. The only way to get this program to work would be to set the length of Text3 before calling the procedure, as shown in example 6.18. Note that as reference mode is the default for all types where it is legal, it is never necessary to give a mode specification for reference parameters. Thus there is no keyword ***reference*** to match *value* and *name*.

**Example 6.18: Concatenate using only reference mode parameters.**

```
begin
  procedure Concatenate(T1, T2, T3); text T1, T2, T3;
  begin
    T3 := T1;
    T3.SetPos(T1.Length + 1);
    while T2.More do T3.PutChar(T2.GetChar);
  end**of**Concatenate**by**reference;
```

```
  text Text1, Text2, Text3;

  Text1 :- "Fred";
  Text2 :- " Smith";
  Text3 :- Blanks(Text1.Length+Text2.Length);
  Concatenate(Text1,Text2,Text3);
  OutText(Text3);
  OutImage
end
```

Name Parameters

Name parameters are very powerful, but complex. It is sometimes possible to make serious errors using them, through failing to consider all possible outcomes of their use. When a variable is passed by name, its use within the procedure has the same effect as when that variable is used outside the procedure. Thus any actions on the parameter inside the procedure directly affect the variable passed in the call. This is obviously a suitable mode for getting values back from a procedure, as we have seen. This contrasts with the use of reference mode, where the contents of what a variable points at are changed, but the variable still points at the same location. If a reference assignment is made to a name parameter, it is actually made to the variable passed originally, not a local copy.

Example 6.15 returned the concatenated texts in the name parameter T3. When the procedure was called, the variable Text3 was passed as this parameter and when the statement following the call was executed, Text3 contained the combined texts. There is one statement missing from this Concatenate. It is needed because the Pos, Length and other attributes of Text3 will be changed by the procedure, when it manipulates T3.
What is this missing line? See if you can work out what it is before reading the polished version below.

Example 6.19 is the version of Concatenate which we can use in all our programs.

**Example 6.19: Finished version of Concatenate.**

```
begin

  procedure Concatenate(T1, T2, T3);
  name T3; text T1, T2, T3;
  begin
    T3 :- Blanks(T1.Length + T2.Length);
    T3 := T1;
    T3.SetPos(T1.Length + 1);
    while T2.More do T3.PutChar(T2.GetChar);
```

```
    T3.SetPos(1); ! Did you get this right?;
  end**of**Concatenate;

  text Text1, Text2, Text3;

  Text1 :- "Fred";
  Text2 :- " Smith";
  Concatenate(Text1,Text2,Text3);
  OutText(Text3);
  OutImage
end
```

The missing statement must reset the position within T3 to the start of the characters it now contains, since it is left pointing to their end. Note that, since name mode is never a default for any type, the mode specifier *name* must be used in a mode specification for any parameters, which are to be used in this way. It is worth mentioning that all parameters passed by name are re-evaluated each time they are used inside the procedure. This is important in some cases since actions inside the procedure may change the value of an expression passed in this way, while expressions passed by value or reference are evaluated and their values copied into the local variables specifying those parameters, once and for all, at the call. Try compiling and running example 6.20 to see the difference. Note also that while it is legal to pass expressions by name in this way, an attempt to assign to a name parameter when an arithmetic expression like those in 6.20 or anything else which is not a valid left hand side has been passed will cause a runtime error. The general rule is that the exact text of what is passed replaces each occurrence of the name parameter within the procedure.

**Example 6.20: Expressions by name and by value.**

```
begin
  procedure Use_Name(Val1, Val2); name Val2; integer Val1, Val2;
  begin
    OuterVal := 3;
    OutInt(Val1,4);
    OutInt(Val2,4);
    OutImage
  end..of..Use_Name;

  integer OuterVal;
  OuterVal := 5;

  Use_Name(OuterVal+3,OuterVal+3)
```

### 6.5.10 File Handling in SIMULA

You are probably used to the fact that computers keep permanent information in collections called files. Some systems use other names such as data sets, but they are essentially the same thing. These files have names by which you can identify them to the computer. Programs can read from these collections of information and write to them.
SIMULA has objects called Files as well. When you want to read from or write to a file on your computer, you must use a SIMULA File object to stand for the external file and tell the computer which external file you want. The exact way that this works may vary slightly from one computer to another, but the important points are the same.
In fact a SIMULA File can stand for any source of or destination for information. A printer can also be written to by using a File object to represent it in your programs. A magnetic tape reader can be used as a source of input in the same way. In fact you have already been using two File objects without being told that that was what you were doing. These are the standard input File, SysIn, and the standard output File, SysOut. Whenever you have used InInt, OutImage and any other input/output instructions you have been using File attributes.

## Simple Input

To read information from the computer we normally use a type of **File object** known as an InFile. In fact InFile is a sub-class(Concept of Class and Sub Class is declared in subsequent sections) of the object type or class called File. This means that all the properties of File are properties of InFile or are redefined in InFile, but that InFile has some extra ones of its own. In fact all types of File objects are sub-classes of File. InFile is not a direct sub-class of File, however; there is another level between them, called ImageFile. Put more simply, class File defines a type of object with a number of attributes used to access sources of and destinations for information on a computer, such as files, printers, terminals and tape readers. File class ImageFile is a sub-class of File. It has all the attributes of its parent class File, some of which it redefines, and in addition some extra attributes used to handle information in certain ways. ImageFile class InFile is a sub-class of ImageFile. It has all the attributes of both File and ImageFile plus extra ones for reading information using the ways suited to ImageFile's attributes. This probably sounds far from simple on first reading, but the idea of thinking of objects as classes and sub-classes is central to SIMULA and so we use it to describe formally the relationships of the various sub-types of File.

Example 6.21 is a program using an InFile to provide its information. Notice the familiar names used for the same purposes, but now prefixed with a File name.

**Example 6.21: Simple input using InFile.**

```
begin

  ref (InFile) Inf;
  text T1;
```

```
    Inf :- new InFile("MYRECORDS");
    Inf.Open(Blanks(80));
    Inf.InImage;
    T :- Inf.Image;
    OutText(T);
    OutImage;
    OutInt(Inf.InInt);
    OutImage;
    Inf.Close
  end
```

There are a few new concepts in this program. Let us look at them one by one. Firstly, we have a new type of declaration. It declares Inf to be a *ref* variable, but has the word InFile in parentheses between the keyword *ref* and the identifier Inf. A *ref* variable is a pointer to an object of a complex type. The class, which defines that type, is given in parentheses after the keyword *ref*. Thus Inf is a location, which can be used to hold a pointer to an object, which is of type InFile. It is initially pointed at an imaginary object called None, just as text variables initially reference NoText.

SIMULA allows us to use complex objects, made up of attributes, which are already defined. These attributes may be of the standard SIMULA types or may reference types defined by the user, i.e. one user defined type may use others as attributes. The construction in SIMULA which can be used to declare a complex type is the class. We have already seen predefined system classes when we looked at File and its sub-classes. Now let us declare a class Lab for use in our program. Example 6.22 shows 6.21 reworked using such a class.

**Example 6.22: Simple labels program with classes.**

```
begin
  integer NumLabs, I;

  procedure OutLine(T); text T;
  begin
    OutText(T);
    OutImage
  end;

  text procedure InLine;
  begin
    InImage;
    inspect SysIn do InLine:- Copy(Image.Strip)
  end;

  class Lab;
  begin
```

```
    text Nam, Street, Town, County, Code;
  end--of--class--Lab;

  ref(Lab) Label1;! Declare a pointer to a Lab object;
  Label1:- new Lab;! Create a Lab object and point Label1 at it;
  comment Remote access through dot notation;
  Label1.Nam:- InLine;
  Label1.Street:- InLine;
  Label1.Town:-InLine;
  Label1.County:- InLine;
  Label1.Code:- InLine;
  InImage;
  NumLabs:= InInt;
  comment Now connected access through inspect;
  inspect Label1 do
  begin
    for I:=1 step 1 until NumLabs do
    begin
      OutLine(Nam);
      OutLine(Street);
      OutLine(Town);
      OutLine(County);
      OutLine(Code)
    end
  end
end
```

Let us look at the new features used here. First there is the class declaration. This provides a description for a class of objects which all have the same attributes. In this case we define Lab (*label* is a SIMULA keyword and may not be used as an identifier). In general, a class declaration is very like a procedure declaration, with the keyword ***class*** instead of *procedure*. The declaration of Lab specifies the name of the complex type being defined as the identifier following the keyword *class*. This identifier is followed by a semi-colon. The attributes of the class are defined in a statement, known as the class body, which follows. Thus Lab has five attributes, all of type text. Having defined the attributes of our new type, we can now create an object, or as many objects as we like, with those attributes. This is done by using an object generator. An object generator can be used as a statement on its own or as a reference expression, i.e. on the right hand side of a reference assignment or as a reference parameter. Examples of all these are shown in 6.23.

**Examples 6.23: Valid occurrences of object generators.**

As a complete statement: new Printer
As the right hand side of a class reference assignment: OutF :- new OutFile
As a class reference parameter: Queue_Up(new Passenger)

A variable is first declared, whose type is *ref*(Lab). This means that it identifies a location where a pointer to an object of the type defined by class Lab may be stored. This variable is used first as the left hand side (destination) of a reference assignment statement. The effect of this statement is that a new object containing the attributes of Lab is created. Since Label1 is assigned a pointer to this object (references it), the object's attributes can be accessed through the variable Label1. As we have seen with objects, which were of types InFile and OutFile, there are two ways of doing this. Both are shown in example 6.20.
**"Remote accessing"** of a class object is done by using the identifier of a reference variable, which currently contains a pointer to the object, Label1 in our example. A *ref*(Lab) procedure could also be used, as we have seen with SysIn and SysOut. This reference is followed by a dot, followed by the name of a visible attribute of the class which defines the type of the object being accessed. This method of accessing attributes may be used for both text objects and class objects. This distinction is important, since the type "text" is not defined by a class. The other way of accessing the attributes of an object is by "connecting" it first. To connect an object we must use an *inspect* statement. In the statement, which follows the keyword *do*, the use of any identifier which has a declaration in the class defining the type of the connected object is assumed to refer to this attribute. If no matching declaration is found in this class or its prefixes, the identifier is assumed to belong outside the object. Thus, within the *inspect* statement in example 6.20, the occurences of Nam, Street, Town, County and Code are taken to refer to attributes of the object Label1, since declarations for them are found in class Lab.

### 6.5.11 Arrays in SIMULA

Many programs need to read, update and write out long series of data items. These items are the objects, which we wish to manipulate. It is rarely worthwhile to use a computer to process one or two items. Even our program, which wrote only a few copies of one label, used an object with a list of data items within it.
The use of files allows us to read lists from outside the program and to store them at its end. Unfortunately, as our updating programs show, it is not a good idea to create a new file, external to the program, each time we add, delete or modify an item in a list. We soon end up with a multitude of out of date files. The use of objects defined by classes allows us to hide a number of basic items inside larger, more complex items. It does not solve the problem of how to refer conveniently to a long list of items in succession. The need to declare and use a separate identifier for each possible line of a letter, for instance, makes long letters unwieldy to process and those of indefinite length almost impossible.
This section is dealing with the first of three problems by using of lists. It provides simple but elegant mechanisms for solving most of the problems mentioned above. Let us start with the problem of holding a long list of items, which are all of the same type.

Example 6.24 shows the use of an array in a much simplified letter program, where no addresses are allowed for, only the text and the name of the sender.

First look at the array declaration in class Letter. (The misspelling is deliberate since there is a system Boolean procedure called Letter, which we might well wish to use in the same program.

**Example 6.24: Letter program using a text array.**

```
begin
  class Leter;
  begin
    text Sender;
    text array Line(1:60);
    integer Len;

    procedure ReadLetter;
    begin
      InImage;
      inspect SysIn do
        while Image.Strip ne ".end" do
        begin
          Len := Len + 1;
          Line(Len) :- Copy(Image.Strip);
          InImage
        end
    end++of++ReadLetter;

    procedure WriteLetter;
    begin
      integer Current;
      for  Current := 1 step 1 until Len do
      begin
        OutText(Line(Current));
        OutImage
      end;
      OutText("          Yours faithfully,");
      OutImage;
      OutText("            ");
      OutText(Sender);
      OutImage
    end++of++WriteLetter;

    OutText("Type your letter, ending with '.end' on a line by itself");
    OutImage;
    ReadLetter;
    OutText("Now type your name on a single line");
    OutImage;
    InImage;
```

```
      inspect SysIn do Sender :- Copy(Image.Strip)

    end--of--class--Leter;

    new Leter.WriteLetter

end**of**program
```

## Simple Array Declarations

The syntax of an array declaration is the type specifier of the items in the list (*integer*, *ref*(Letter) etc.), followed by the keyword *array*, followed by an identifier, followed by the "bounds" of the list, enclosed in parentheses. Spaces (or ends of line) are used to separate keywords and identifiers as usual. They are not required between the identifier and the left parenthesis, but may be used if you wish. It is legal to omit the type specifier, in which case the array is assumed to be of type real. The syntax has not included the form of the bounds. In the commonest case we wish to declare a simple numbered list. The bounds then are two arithmetic values, which are converted to integers if necessary, separated by a colon. In example 6.24 the constant integers 1 and 60 are the bounds. This definition is only the simplest variant, but it covers most uses of arrays for the moment.
The semantics of such a declaration produce information telling the system to reserve space for a list of items of the specified type. This list is to be numbered consecutively, starting with the value before the colon and ending with the value after the colon. This also defines the number of elements in the list. This list as a whole is referred to by its identifier. Thus a whole array can be passed as a parameter to a class or procedure, by giving just the identifier. Individual items in the list can be referred to by the identifier followed by an arithmetic value enclosed in parentheses, giving the number of the element to be accessed, within the list. Thus the declaration in example 6.24 tells the SIMULA system to reserve space for a list of sixty text variables. These are to be declared to be numbered from one to sixty. The list will be referred to in the program by the identifier Line.
Note that the value of the first bound does not have to be 1. The bounds can have any values, even negative ones, as long as the first bound is less than the second or equal to it. The first bound is usually referred to as the lower bound and the second as the upper.Note also that the values of the bounds may be given as real values. In this case they are converted to integers in the same way as for assignments. The values can be arithmetic expressions as well as constants. The normal rules for evaluating expressions apply.

## Using Array Elements

The items in an array list are often called its "elements". Example 6.24 shows how an individual element of Line can be accessed. This is known as a subscripted variable. The value within the parentheses is called the subscript or the index. Item number Len of the list is accessed in ReadLetter. It is referred to as Line(Len). Since Len is increased by one

before each Image is copied to Line(Len) the effect is to copy successive lines of input into successive elements of the text array Line. The syntax of a simple subscripted variable is an identifier followed by an arithmetic value enclosed in parentheses. The arithmetic value may be a constant, a variable or a more complicated expression, including a call on an arithmetic type procedure. Where necessary the value will be converted to an integer, following the normal rules. The semantics are also simple. The value of the subscript gives the number used as an index to the elements of the array. Note that the value of the lower bound is important in determining which element this refers to. A subscript of six will only refer to the sixth element if the lower bound was one. If the lower bound was four, indexing by six gives the third element.
A subscripted variable may be used wherever a simple variable of the same type may be used. The value of the subscript, converted to an integer if necessary, must lie between the values of the lower and upper bounds, inclusive. If it is out of this range a runtime error will be reported.

## Dynamic Array

Clearly the use of arrays allows large amounts of data to be held in locations declared within our programs, without the continual need to access files and without declaring long lists of identifiers. The use of loops, especially ***for* loops**, allows us to handle arrays in concise and clear ways. One problem with the use of arrays is that we must tell the system in their declarations how many elements they contain and what their bounds are. Often this may not be known until runtime. This means that example 6.24 can only cope with letters of up to sixty lines. If someone wanted to use the program for a longer letter, they would have to alter the source and recompile it. Although the array is not always the best solution when there is no way of knowing in advance how long the list will be, it can be made more generally useful by specifying the bounds in other ways. We have defined the bounds as any expressions giving arithmetic values. This includes constants, as used in 6.24, but also variables and expressions involving operators and type procedures. The only restriction is that any variables used must already have their values fixed before entering the block in which the array is declared. This means that the bounds can changed each time a block is entered.
The SIMULA system allocates the space used by each block only when that block is entered. Thus it does not need to know how big an array is until then. If a variable used in the bounds of an array has been declared in an outer block, this variable can have its value set in that outer block before the array's space is allocated in the inner block. The variable must not be declared in the same block as the array, since the system may allocate the block's arrays before its other variables and, anyway, these variables could only have their initial zero values, since no statements may come in front of declarations in a block. As a consequence, the only block which cannot use variables in array bounds is the program block. This is the outermost block and must use constants in all its array bounds. All sub-blocks, procedures and classes used in a program are free to use variables in array bounds so long as these are declared in an enclosing block or, for classes and procedures, are parameters. Remotely accessed variables may also be used.

The simple examples in 6.25, 6.26 and 6.27 show how "dynamic bounds", as this mechanism is known, may be used for sub-blocks, procedures and classes respectively. These trivial examples demonstrate a very powerful facility. One important point to note is that when the parameters of a procedure or class are used in bounds for arrays declared in that procedure or class body, they are treated as outside that body. This is the only case where any distinction is made between parameters and other locally declared variables inside the procedure or class body. It is very important that this be allowed.

**Example 6.25: Dynamic array bounds in a sub-block.**

```
begin
  integer I1,I2;   ! Declared at the outermost block level;
  I1 := 2;         ! Sets a non-zero value in I1;
  I2 := 3;         ! Sets a non-zero value in I2;
  begin

    comment Start a sub-block which can only be entered after
          I1 and I2 have had their values set;

    integer array A1(I1:I2);   ! Declare with I1 and I2 as bounds;
    A1(2) := 6;
  end--of--sub-block;
  comment Array no longer accessible;
end
```

**Example 6.26: Dynamic array bounds in a procedure.**

```
begin

  procedure Bounder(Lowest); integer Lowest;
  begin
    comment Parameters may be used as bounds inside a procedure body;

    character array C1(Lowest:4*Lowest); ! Use an expression containing
                                         Lowest as upper bound;
    C1(Lowest*2+1) := '&';   ! Use an expression in the subscript too;
    OutChar(C1(5));          ! Null unless Lowest is 2;
    OutImage
  end--of--procedure--Bounder;

  Bounder(2);                ! Should print &;
end**of**program
```

Example 6.27: Dynamic array bounds in a class.

```
begin
      integer Lower;
      class Cl1(Upper); integer Upper;
      begin
        Boolean array BoolArr(Lower:Upper); ! Use a mixture of enclosing
                          block's declarations and parameters to set
                               bounds;
        BoolArr(Lower+3) := True
      end--of--class--Cl1;

      ref(Cl1) Cl1Ref;
      Lower := 4;      ! Sets lower bound before object generation;
      Cl1Ref :- new Cl1(7);      ! Passes upper bound as a parameter;
      if Cl1Ref.BoolArr(5) then OutText("True") else OutText("False");
      OutImage
    end**of**program
```

### 6.5.12 Sub-Classes (Inheritance)

One of the important ways we have of making sense of the world is to classify things. We put them into categories or classes. SIMULA allows us to reflect this very natural way of thinking in the way we write programs. When classifying things we first group them either very generally, e.g. as animal, vegetable or mineral, or very specifically, e.g. as bees or roses, depending on circumstances. These approaches correspond to the programming techniques known as "top down" and "bottom up" design, respectively. In practice, it is fairly easy to classify things in general terms, but appearances can be deceptive when it comes to detail. Things, which look alike, may actually have very different origins. Thus the hedgehog and the spiny anteater look remarkably similar and live very similar lives, yet, genetically, they are not closely related at all. SIMULA takes the top down approach as the safest, just as natural science has tended to. It allows us to define a CLASS, as we have seen, to represent a general type of object. This may then be extended, to reflect the special characteristics of sub-types by defining sub-classes of the original. These retain some or all of the characteristics of the parent type, but include characteristics, which are only found in certain objects of this type. It is important to notice that sub-types in SIMULA extend and refine the range of characteristics of the parent type. The more general the class of objects described, the fewer characteristics that are given to it. One example of the use of such sub-types, that we have already seen, is the Class File and its sub-classes.

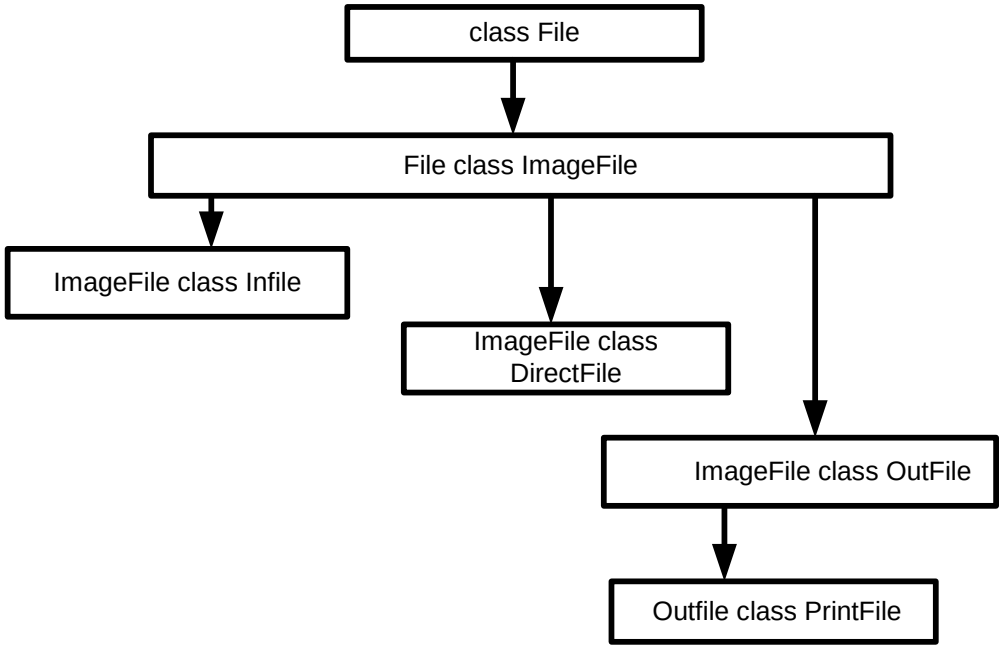



**Figure 1 : Family tree of class File**

The syntax of a sub-class declaration is very simple. The keyword *class* is preceded by the name of the parent class. Otherwise the declaration is the same as for a simple class. The new class is said to be "prefixed" by the parent.

The Types of Class Objects

When a class object is generated it possesses all the attributes of the class whose name is given in the object generator. This includes any visible attributes from classes on its prefix chain, following the rules given above concerning name clashes. The type of such an object is the class specified and this is called its qualification. It can also be thought of as being qualified by the classes on its prefix chain, except that not all the attributes of these may be visible. A variable which is declared as a *ref* to a class which is the qualification of an object or is on the prefix chain of its qualifying class may be used to access that object. The type of the refIt is only legal to treat an object which is being remotely accessed as if it was qualified by the class of the referencing variable. Reference variable used controls how much of the prefix chain may be so accessed.

## 6.6 Case Study I

A multi-channel queuing system-Active Customer approach

Figure 2 shows the system being simulated. It is an abstraction of for example a bank where the customers wait in one queue for any of the tellers. The interval between arrivals is random, uniformly distributed between 1 to 3 minutes. All servers have the

same random service time that is normally distributed with the mean value 8 minutes and the standard deviation 2 minutes. The simulation should find the average time a customer spends in the system. Simulation of similar systems in Simula (exactly in the system class Simulation of Simula) always starts by identification of processes. One process is obviously the generator of customers - it will repeatedly generate a customer, record its arrival time, and wait a random delay. To express the dynamics of the system, there are two logical approaches. First (used in this example) is based on active customers and passive servers. The opposite approach - active servers, passive customers is shown in the next example. An active customer has life rules represented by the following steps:
If there is a free server, proceed. Wait in the queue otherwise. Seize a server; generate random delay that represents the service time. Release the server.
If there is a waiting customer (if the queue is not empty), remove it from the queue and activate it. (The activated customer will start its step 2.)
Update statistics.

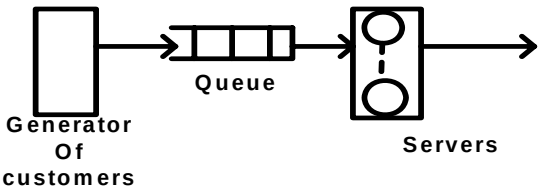



Figure 2: Queuing system made of one queue and more servers.

The following program is a simulation model of the above system. Note, that the primary objective was to show the logic of the model in a program as simple as possible. Real simulation models of course prompt for all variable parameters and provide more results (like for example average and maximum queue length, etc). In the following program there are two processes that exist during the whole experiment: the generator and the main program (block prefixed by Simulation), that just waits until the experiment is over and then displays the result. Then there are a varying number of customer processes that exist temporarily. After updating statistics the customers terminate. Note the use of standard functions to generate random delays. There are standard functions in Simula for most commonly used distributions. All are given an integer variable as a seed to be used by the random generator. So all random values may use separate streams of random numbers or share a common one.

```
! Active customer approach;
Simulation Begin
        Real  TrialDuration;                    ! Experiment length [min];
Ref(Head)  Queue;                               ! The queue;
Integer  Servers;                               ! Total number of servers;
Integer  BusyServers;                   ! Numbers of working servers;
        Integer  TrialSeedG, TrialSeedS;            ! Seeds of random generators;
        Long Real  TotalTime, TimeSpent;            ! Variables for statistics;
Integer  CustomersOut;                          ! Number of served customers;
        Real  MinInt, MaxInt;                       ! Uniform interval between arrivals;
Real  SMean, SStd;                              ! Normal service duration;
Process Class Generator;
```

```
Begin
While true do begin
Activate New Customer(Time);        ! Time is the current (arrival) time;
! Interval between arrivals: ;
Hold(Uniform(MinInt, MaxInt, TrialSeedG));
End While;
End of Generator;
Process Class Customer(Arrival);  Real Arrival;
Begin
Ref(Customer)  Next;
        If not Queue.Empty or (BusyServers >= Servers) then
Wait(Queue);                                ! Customer has to wait in the Queue;
                                                    ! Service can start: ;
BusyServers := BusyServers + 1;             ! Seize a server;
! This is the teller service: ;
Hold(Normal(SMean, SStd, TrialSeedS));
BusyServers := BusyServers - 1;             ! Release the server;
If not Queue.Empty then begin
Next :- Queue.First;
Next.Out;                                   ! First from Queue served;
Activate Next after Current;
End If;
CustomersOut := CustomersOut + 1;           ! Statistics;
TotalTime := TotalTime + (Time - Arrival);
  End of Customer;
 ! MAIN program body: ;
  TrialSeedG := 7;  TrialSeedS := 23;       ! Seeds for random variables;
  MinInt := 1; MaxInt := 3;                 ! Min and Max intervals;
  SMean := 8; SStd  := 2;                   ! Random normal servers;
  OutText("Enter the number of Servers : "); OutImage;
  Servers := InInt;                         ! Initial numbers;
  TrialDuration := 600;                     ! Other variables initialized to 0;
  Queue :- New Head;                        ! Create an empty queue;
  Activate New Generator;                   ! This starts the experiment;
  Hold(TrialDuration);                      ! Experiment duration;
  TimeSpent := TotalTime/CustomersOut;
  OutText("Average time spent in the system: ");
  OutFix(TimeSpent, 3, 10); OutImage;
  InImage
End of program;
```

## 6.7 Case Study II

A multi-channel queuing system-Active Server approach

The system being simulated is the same, as in the previous example (refer Figure 2). The difference is the active server that repeatedly serves customers from the queue until the queue is empty. Then the server passivate . Customers at first activate all idle servers (if any) and then go into the queue. This is of course not very efficient, but simple. Servers activate customers after completing the service. In the rest of their lives the customers just update statistics. The main program creates and activates all servers, but they immediately passivate, because the queue is empty. Then the main program activates the generator and waits until the experiment is over.

```
 ! Active server approach;
Simulation Begin
        Real  TrialDuration;                    ! Experiment length [min];
Ref(Head)  Queue;                               ! The queue;
Integer  Servers;                               ! Total number of servers;
Integer  TrialSeedG, TrialSeedS;      ! Seeds of random generators;
Long Real  TotalTime, TimeSpent;                ! Variables for statistics;
Integer  CustomersOut;                          ! Number of served customers;
Real  MinInt, MaxInt;                           ! Uniform interval between arrivals;
Real  SMean, SStd;                              ! Normal service duration;
Ref(Server) Array ServBank(1:10);   ! Max. Number of servers is 10;
Integer i;
Process Class Generator;
Begin
While true do begin
Activate New Customer(Time);
! Interval between arrivals: ;
Hold(Uniform(MinInt, MaxInt, TrialSeedG));
End While;
  End of Generator;
  Process Class Server;
  Begin
Ref(Customer) ServedOne;
        While true do
If not Queue.Empty then begin
      ServedOne :- Queue.First;
      ServedOne.Out;                    ! First from Queue served;
      Hold(Normal(SMean, SStd, TrialSeedS));
      Activate ServedOne after Current
end
Else       Passivate;
  End of Server;
  Process Class Customer(Arrival);  Real Arrival;
  Begin
For i:=1 step 1 until Servers do
If ServBank(i).Idle then
Activate ServBank(i) after Current;
```

```
Wait(Queue);
        ! Service finished: ;
 CustomersOut := CustomersOut + 1;  ! Statistics;
 TotalTime := TotalTime + Time - Arrival;
 End of Customer;
         ! MAIN program body: ;
  TrialSeedG := 7;  TrialSeedS := 23;          ! Seeds for random variables;
  MinInt := 1; MaxInt := 3;                    ! Min and Max intervals;
  SMean := 8; SStd  := 2;                      ! Random normal servers;
  OutText("Enter the number of Servers : "); OutImage;
  Servers := InInt;                            ! Initial numbers;
  TrialDuration := 600;
  Queue :- New Head;
  For i:=1 step 1 until Servers do begin
    ServBank(i) :- New Server;
   Activate ServBank(i)                ! Create and activate all servers;
  End For;
  Activate New Generator;                      ! This starts the experiment;
  Hold(TrialDuration);                         ! Experiment duration;
  TimeSpent := TotalTime / CustomersOut;
  OutText("Average time spent in the system: ");
  OutFix(TimeSpent, 3, 10); OutImage;
  InImage
End of program;
```

## 6.8 Disadvantage of SIMULA

SIMULA never became a widely spread commonly used language. There are various reasons to explain this fact. Even though the reasons all depend on each other, the following is an attempt to group them from various points of view.

- Born in a small European country
- Expensive
- Does not have a modern Interactive development Environment (IDE)
- Too complicated
- Not enough publications
- Limited file access facilities (typed files)
- Missing data types (records, sets)
- No advanced parallelism and real time support
- No GUI support
- Long executable files for short programs
- No multiple inheritance
- No interfaces
- No automatic collection of statistics
- No report generator

## 6.9 Summary

In this unit we have learn different kinds of simulation languages. SIMULA is studied in detailed .SIMULA is first object oriented programming language , which having all OOPS features like concepts of encapsulation, data hiding, classes, objects and inheritance together the features of any high level language i.e. procedures, loops conditional statements etc. SIMULA's syntax and semantics is almost similar to other high level object oriented programming language i.e. C++ and Java .At the end of unit we have discussed two case studies which will be helpful to apply the reader's knowledge.

## 6.10 Key Words

Continuous simulation languages, discrete simulation languages, Block oriented simulation languages, Expression oriented continuous languages, Flowchart oriented languages, Activity oriented languages, Event oriented languages, Process oriented languages SIMULA, SIMULA I, SIMULA 67, SIMULA Data types, SIMULA Statements, SIMULA Procedures, SIMULA classes, Nested classes, SIMULA STANDARD PACKAGES, QP.

## 6.11 Self-Assessment Questions

Q.1 What is the Simulation language? Discuss the different kinds of simulation language.

Q.2 Flow chart simulation language comes in which category? Give an example of it.

Q.3 Describe the discrete and continuous simulation language with one daily life example.

Q.4 In each of the following program blocks, find and correct the syntax errors.

a)
```
begin
 integer I1.I2;
  I1 := 3;
  I2 := I1
  OutInt(I2,4);
  OutImage
```

```
    end
```

b)

```
   begin
      Res := 4;
      integer Res;
      OutInt(Res,6);
      OutImage
    end
```

Q.5 Correct the following programs.

a)

```
  begin
    integer IntVal;
    IntVal := '3'
 end
```

b)

```
  begin
    character LETTER;
    LETTER := "J"
 end

 begin
    OutInt("34",2);
    OutImage
 end
```

Q.6 A computer fault has changed some small letter a's into ampersands, '&'. Write a program to scan a text and correct this.

Q.7 A similar fault has changed every occurrence of the word "and" to the word "boe". Write a program to correct texts, which have suffered this fate.

Q.8 Use a recursive procedure to write a program which scans a text for a occurrences of a sequence of characters and replaces them with another.

Q.9 Write a program which reads the names of a group of students and creates an array of pupil records, holding name, age, address and marks in math, English and physics. Allow student details to be filled in any order, copying the details into the correct entry in the array. Print out the contents in the order in which the list of names was first given.

## 6.9 References/ Suggested Readings

1. Kirkerud, B., “Object-Oriented Programming with SIMULA”, Addison-Wesley
2. Pooley, R.J., “An Introduction to Programming in SIMULA”, Oxford, Blackwell Scientific Publications
3. N. Deo , “ System Simulation”,PHI.

**Subject : System Simulation and Modeling** **Author : Jagat Kumar**
**Paper Code: MCA 504** **Vetter : Dr. Pradeep Bhatia**
**Lesson : Use Of Database, A.I. In Modeling And Simulation**
**Lesson No. : 07**

## Structure

## 7.0 Objective

Data Base and Artificial Intelligence having their wide applications in almost all branches in Science and Engineering, so these subjects having their impact in modeling and simulation also. In this unit you will learn how database served as input to defined model and output data is stored in database to further analysis. The application of Artificial Intelligence (A.I.) concepts like Fuzzy Sets to solve complex partial differential equations, and application of neural networks in modeling of complex system is also discussed in this unit.

## 7.1 Introduction

Database mainly used in modeling and simulation for a complex system modeling. A complex system requires a large amount of input data and simultaneously produce large output data for analysis purpose.

How can Simulation and A.I. (Artificial Intelligence) be used to help invent these new technologies and improve the use of the existing ones? in fact, they already are in used . Team-training simulators have had a considerable impact on the tactical performance of military and emergency response personnel, and wargaming simulations are routinely used to assist strategic planning by their leaders. Simulation and A.I. are also being used to assist contingency planning for military operations and their logistic support. There would seem to be many other ways in which AI could be used to expand and enhance the benefits of simulation both in training and in the execution of system in which decision is required. One of the most promising may be one that at first seems the most improbable and counter-intuitive – the use of A.I. as a tool for innovation.
However, it may be possible to use genetic algorithms, or a similar evolutionary technique, to build scenarios with problems that we have not yet considered. To get started, we would need to break up our existing threat scenarios into component parts and reassemble these parts, randomly or systematically, into new candidate scenarios without regard for their viability or perceived usefulness. The algorithm would test these candidates against our current or planned defenses, and evolve ever more dangerous scenarios by combining parts from the most fit, where “fitness” was defined as ability to overcome our defenses. Modeling and simulation would be used to test the candidate scenarios against those defenses and to evaluate the fitness of each scenario.

## 7.2.Database in Modeling and Simulation

The objective of the database and its data model in simulation is to provide a flexible representation of data for conducting computational simulations. The simulation data model is closely related to the data model for experimental testing .A data model is a specific description of the type of information and relationships between the information needed for a problem domain. For computational simulation, the simulation data model encompasses input data needed to define a simulation model and computational procedure, the output data from a simulation, and information about the simulation software and its execution that transformed the input data to the output data. To enable searching and querying in the data repository, the data model includes information that will be useful for a wide variety of queries. Generally the data model for simulation has been developed to allow development of simulation applications within independently of any specific simulation program.

The modeling of data is a critical activity for the database or repository to meet the needs of applications and users . The term *data model* was first introduced by the Codd in 1980 to include:
1. A collection of data object types that form the basic building blocks for a database system that defines the information of a problem domain.
2. Rules for define the constraints on the data in the objects.

3. Operators that can be applied to the objects for retrieving data or other purposes.

Data models and the process of data modeling have become an essential first step in designing databases and repositories. Most data modeling is based on the concept of entities and relations, which were first developed as the so-called entity-relationship (ER) model. Entities are types of information of data and relationships represent the associations between entities. The modern approaches for data modeling use the concepts of object-oriented design in which entities are represented as classes, which are templates for the software behavior of the objects generated by a class. A class has a name, attributes, constraints, and relationships with objects of other classes. In an object-oriented sense, classes define valid operations on objects of the class. Relationships between classes may be associations with cardinality, an inheritance relationship, or other advanced relationships such as aggregation or composition.
The object-oriented approach is used for developing the high-level view of the simulation data model.

### 7.2.1 Definition of the Simulation Data Model

In defining a data model for simulation applications, the first decision was on the granularity of the data to be represented. The decision is primarily based on what are the most important user needs for querying, searching, and accessing computational simulation information. The decision was made to use a high-level view as interrelated files used in a simulation, along with descriptive information about the content of the files. This approach allows users to search for files for simulations that meet search criteria about the problem, software, project, and other high-level attributes. An alternative would be to provide a representation of the structural or geotechnical systems being analyzed. This latter approach would require a very detailed data on all aspects of a simulation including element types, materials, geometry, constraints, etc. It is not expected that many users would browse or query a database for simulation using search criteria that requires such specific information. The difficulty of the development of a detailed data model and implementation in a database system would outweigh the benefits to users.
Figure 1 is a basic view of the database required for simulation. The primary concepts are a simulation event, input files that may be parameterized, and output files produced by a software program.

There is a tradeoff between removing redundant data from the database. The functional relationships between the data and efficiency in the implementation of a database system. Redundant data or relationships should not be represented because such data are difficult to update and can lead to inconsistent information. Normalization theory is used to analyze the functional relationships and reduce redundancy. However, highly normalized data requires joins between data types, which may be inefficient unless the database system has very efficient query optimization. For the data model for simulation, redundancies are avoided, and most of the information is in at least third normal form.
An important aspect of a data model is constraints, which can take the form of valid ranges or values of attributes, cardinality, or on rules governing relationships. Database systems have varying capabilities for defining and enforcing constraints specified in a data model. In general, the data model for simulation has only the essential constraints needed for the integrity of the data. In the following subsections, the classes in the data model and their relationships are described. The assist in the definition, the description proceeds from simple classes to the more complicated classes.

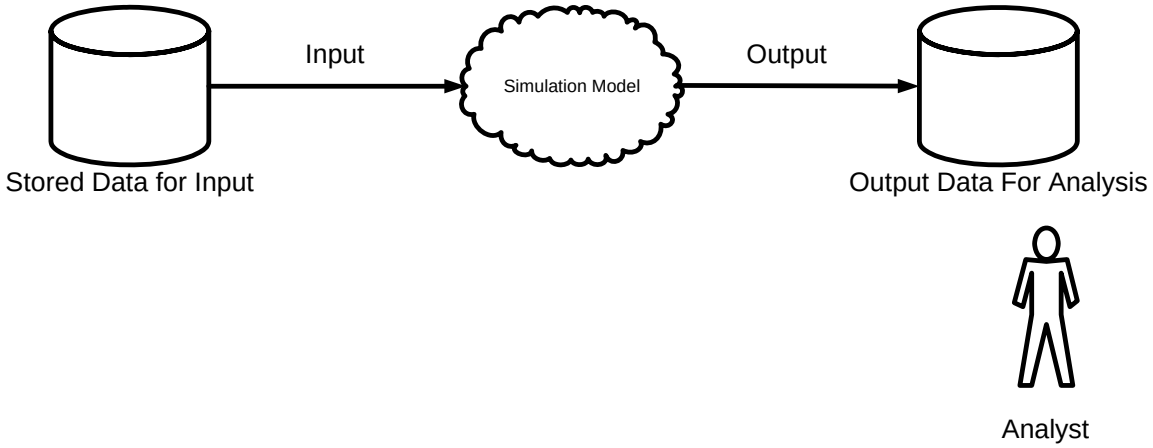



**Figure 1:A basic database concepts for Simulation**

### *7.2.2 Data Representation for a Simulation Event*

Using the notion of an *event* as defined in the data model for experiments, Simulation Event has attributes describing the event and time stamp information. Simulation Event is represents the execution of a specific computational simulation using a set of input data and producing a set of output data as computed on a computational resource. A simulation event is conducted by a Person for a Project. Simulation is executed on a computer hardware and software. The input for a Simulation Event is represented by an association with an Input File Parameter Set. The results of the simulation are stored in multiple files associated with Data Files.

### *7.2.3 Data Representation for Input Files for a Simulation*

A key concept in conducting a computational simulation is that a problem may have a number of parameters and a specific simulation can be executed with a defined set of parameter values. This is represented in the data file by allowing input files to have parameter sets associated with them.
Input File is the top-level file that defines the input for a simulation. The input file is associated with a Person, Project. Since most models are defined with a number of files (such as using a source or include command), the data model represents the referenced files by an association with a Data File.

### *7.2.4 Data Representation for Output Files for a Simulation*

A Simulation Event produces a number of output files, of which each is represented as a Data File. A Data File can be associated with two files, one representing the numerical values and the second containing descriptive information (such as an XML description) for the contents of the numerical file. Each type of file has a name, description, and a universal locator.

## 7.3 A.I. in Modeling and Simulation

A.I. has an equally valuable role to play in avoiding misunderstandings between human and machine so that the two can be real partners in planning not only for efficient and safe routine operations, but also for recovering from unavoidable faults in simulation.

A.I. techniques, such as evolutionary algorithms and influence diagrams, can be used to detect kind of incremental degradation in the automated process of simulation before it rises up to bite us. However, its correction depends on human awareness. Considerably more creativity is needed for intelligent adaptation to the truly unexpected equipment failures or sudden changes in the external environment.

The planning system by which the operator and the software interact needs to include, or even emphasize, contingency planning. Keeping risk under control requires analysis both of what can go wrong (for instance, transient disturbances and component failures) and of what strategies are available to reduce the likelihood of disturbances or mitigate their effects. The system should make plans against a variety of high-risk contingencies, but it is an intractable problem to compute all possible contingencies for all possible states of even a simple system, let alone one that is worth automating.
Instead, starting with the current state of the plant and the next planned transition, genetic algorithms can use the system's concurrent simulation to play the "devil's advocate" by finding the worst-case scenarios and their likelihood.
The planning system should follow the same rule as medical treatment: "First, do no harm." Planning for recovery is further complicated by the fact that most real systems are not Markovian their current state does not tell us everything we need to know about them.

AI techniques such as procedural heuristics and constraint resolution can be used to guide the simulation in deciding when and if automatic protective devices can be overridden, in planning restart and workaround, As recovery proceeds, planning should always be working toward optimization (using evolutionary algorithms or other hill-climbing techniques), while engaging in a continual trade-off with the demands of reliable operation. These calculations generate a set of Pareto-optimum choices for each level of risk. The choices form a surface in the space of system parameters. Although this space has very high dimensionality, only those dimensions need be displayed that represent parameters directly controllable by the operator. Despite all that AI and simulation can do, human creativity may be the best source of adaptation to the unexpected. The planning system needs to take advantage of that while still protecting the automated plant from human error. The human needs to be warned, and sometimes even forcibly stopped, from taking actions that, based on the concurrent simulation, are detrimental, or downright dangerous. Mixed initiative human/machine planning in real time finally comes down to making the human aware of the total situation without information overload. Warnings and alerts are only a part of that, but they are a large part. Early warnings are needed and prediction will always be less than perfect. One possibility is to issue probabilistic warnings or "fuzzy alerts." These may be satisfactory for suggesting to

the user where to look for problems and with what priority, but not for alarms that require immediate action. It is not clear how much we can expect of pre-attentive processing on the part of the operator. A small change in the display is too easy to ignore, but the operator will also learn to ignore a whole series of alarms if their threat is seldom realized. The precision and reliability of the warnings require good design practice and careful implementation of the concurrent simulation and any A.I. methods that are employed.

The main current real use of artificial intelligence methods in social science for theoretical work is the development of explicit theoretical models of cognitive processes. For Cognitive Sciences, theoretical based simulations are essential. Computational cognitive science is directly relevant for sociological work: For example as a theory basis of data collection, for the theory of personal identity, emotions and for the theory of interaction structures. Data structures like frames and scripts can be directly imported and included as part of the working model of the actors of a sociological simulation. Therefore, the possible applications of artificial intelligence for theory construction in the social sciences may be more in the domain of using data structures of A.I. ,simulations for the organization of every day knowledge required by individual actors.

The power and usefulness of artificial neural networks have been demonstrated in several applications including speech synthesis, diagnostic problems, medicine, business and finance, robotic control, signal processing, computer vision and many other problems that fall under the category of pattern recognition. For some application areas, neural models show promise in achieving human-like performance over more traditional artificial intelligence techniques.

### 7.3.1 Neural Network in Modelling and Simulation

What, then, are neural networks? And what can they be used for? Although von-Neumann-architecture computers are much faster than humansin numerical computation, humans are still far better at carrying out low-level tasks such as speech and image recognition. This is due in part to the massive parallelism employed by the brain, which makes it easier to solve problems with simultaneous constraints. It is with this type of problem that traditional artificial intelligence techniques have had limited success. The field of neural networks, however, looks at a variety of models with a structure roughly analogous to that of the set of neurons in the human brain.

The branch of artificial intelligence called neural networks dates back to the 1940s, when McCulloch and Pitts developed the first neural model. This was followed in 1962 by the *perceptron* model, devised by Rosenblatt, which generated much interest because of its ability to solve some simple pattern classification problems. This interest started to fade in 1969 when Minsky and Papert provided mathematical proofs of the limitations of the perceptron and pointed out its weakness in computation. In particular, it is incapable of solving the classic exclusive-or (XOR) problem. Such drawbacks led to the temporary decline of the field of neural networks.

The last decade, however, has seen renewed interest in neural netivorks, both among researchers and in areas of application. The development of more-powerful networks,

better training algorithms, and improved hardware have all contributed to the revival of the field. Neural-network paradigms in recent years include the Boltzmann machine, Hopfield's network, Kohonen's network, Rumelhart's competitive learning model, Fukushima's model, and Carpenter and Grossberg's Adaptive Resonance Theory model . The field has generated interest from researchers in such diverse areas as engineering, computer science, psychology, neuroscience, physics, and mathematics.

An Artificial Neural Network is a network of many very simple processors ("units"), each possibly having a (small amount of) local memory. The units are connected by unidirectional communication channels ("connections"), which carry numeric (as opposed to symbolic) data. The units operate only on their local data and on the inputs they receive via the connections.

The design motivation is what distinguishes neural networks from other mathematical techniques: A neural network is a processing device, either an **algorithm**, or **actual hardware**, whose design was motivated by the design and functioning of human brains and components thereof.

There are many different types of Neural Networks, each of which has different strengths particular to their applications. The abilities of different networks can be related to their structure, dynamics and learning methods.
Neural Networks offer improved performance over conventional technologies in areas which includes: Machine Vision, Robust Pattern Detection, Signal Filtering, Virtual Reality, Data Segmentation, Data Compression, Data Mining, Text Mining, Artificial Life, Adaptive Control, Optimization and Scheduling, Complex Mapping and more.
Neural networks have been applied to a wide variety of different areas including speech synthesis, pattern recognition, diagnostic problems, medical illnesses, robotic control and computer vision.
Neural networks have been shown to be particularly useful in solving problems where traditional artificial intelligence techniques involving symbolic methods have failed or proved inefficient. Such networks have shown promise in problems involving low-level tasks that are computationally intensive, including vision, speech recognition, and many other problems that fall under the category of pattern recognition. Neural networks, with their massive parallelism, can provide the computing power needed for these problems. A major shortcoming of neural networks lies in the long training times that they require, particularly when many layers are used. Hardware advances should diminish these limitations, and neural-network-based systems will become greater complements to conventional computing systems.
Researchers at Ford Motor Company are developing a neural-network system that diagnoses engine malfunctions. While an experienced technician can analyze engine malfunction given a set of data, it is extremely complicated to design a rule-based expert system to do the same diagnosis. Marko et al. trained a neural net to diagnose engine malfunction, given a number of different faulty states of an engine such as open plug, broken manifold, etc. The trained network had a high rate of correct diagnoses. Neural nets have also been used in the banking industry, for example, in the evaluation of credit card applications.

Most neural network applications, however, have been concentrated in the area of pattern recognition, where traditional algorithmic approaches have been ineffective. Such nets have been used for classifying a given input into one of a number of categories and have demonstrated success, even with noisy input, when compared to other more conventional techniques.

Since the 1970s, work has been done on monitoring the Space Shuttle Main Engine (SSME), involving the development of an Integrated Diagnostic System (IDS). The IDS is a hierarchical multilevel system, which integrates various fault detection algorithms to provide a monitoring system that works for all stages of operation of the SSME. Three fault-detection algorithms have been used, depending on the SSME sensor data. These employ statistical methods that have a high computational complexity and a low degree of reliability, particularly in the presence of noise. Systems based on neural networks offer promise for a fast and reliable real-time system to help overcome these difficulties. Neural networks in this application allow for better performance and for the diagnosis to be accomplished in real time. Furthermore, because of the parallel structure of neural networks, better performance is realized by parallel algorithms running on parallel architectures.

At Boeing Aircraft Company, researchers have been developing a neural network to identify aircraft parts that have already been designed and manufactured, in efforts to help them with the production of new parts. Given a new design, the system attempts to identify a previously designed part that resembles the new one. If one is found, it may be able to be modified to conform to the new specifications, thus saving time and money in the manufacturing process.

Neural networks have also been used in biomedical research, which often involves the analysis and classification of an experiment's outcomes. Traditional techniques include the linear discriminant function and the analysis of covariance. The outcome of the experiment is in some cases dependent on a number of variables, with the dependence usually a nonlinear function that is not known. Such problems can, in many cases, be managed by neural networks.

Stubbs presents three biomedical applications in which neural networks have been used, one of which involves **drug design**. Nonsteroidal antiinflammatory drugs (NOSAIDs) are a commonly prescribed class of drugs, which in some cases may cause adverse reactions. The rate of adverse reactions (ADR) is about 10%, with 1% of these involving serious cases and 0.1% being fatal. A three-layer backpropagation neural network was developed to predict the frequency of serious ADR cases for 17 particular NOSAIDs, using four inputs, each representing a particular property of the drugs. The predicted rates given by the model matched within 5% the observed rates, a much better performance than by other techniques. Such a neural network might be used to predict the ADR rate for new drugs, as well as to determine the properties that tend to make for "safe" drugs.

### 7.3.2 Fuzzy Sets in Modelling and Simulation

We have discussed continuous systems in unit V in that we have already learnt the process of evolution depends on differential equations. Such a system contains a number of parameters that must be estimated (for instance, the a, b, c, $d > 0$ in the predator-prey model, discussed in unit V). Usually point estimates are calculated and used in the model. These estimates typically have uncertainty associated with them.

We can incorporate uncertainty in our differential equations. This is done by using fuzzy numbers as estimates of the unknown parameters.

**Fuzzy Sets (1/3)**

In a classical set, an element is either a member of the set or not. Fuzzy sets are defined in terms of classical sets.
**Definition:** A fuzzy set A on a classical set X is defined as:

$$A = \{(x, \mu_A(x)) \mid x \in X\}$$

The membership function $\mu_A(x)$ quantifies the grade of membership of the elements x of the fundamental set X. For the functional values of $\mu_A(x)$, we have the following properties

$$\forall_{x \in X} \mu_A(x) \geq 0$$
$$\sup_{x \in X} \{\mu_A(x)\} = 1$$

**Fuzzy Sets (2/3)**

If the fuzzy set B is defined as B = {(3, 0.3), (4, 0.7), (5, 1), (6, 0.4)}, it is a standard fuzzy notation to write
B = {0.3/3, 0.7/4, 1/5, 0.4/6}
Any value with a membership grade of zero does not appear in the expression of the set.
Classical sets are called **crisp sets**, to distinguish between them and fuzzy sets.

**Fuzzy Sets (3/3)**

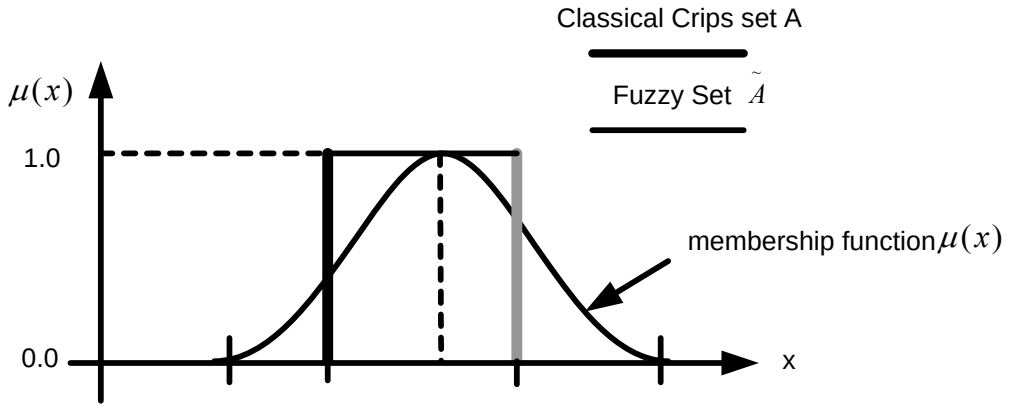



**Figure 2: Relationship Between Fuzzy Set and Classical Crisp Set**

Fuzzy sets are used in fuzzy logic, which is an extension of Multi-valued logic. This is used for "approximated reasoning". Apart from that, fuzzy sets also form the basis for fuzzy numbers.

**Fuzzy Numbers**

A fuzzy number is a convex, normalized fuzzy set $A \subseteq R$, where R is a set of Real numbers, whose membership function is at least segmentally continuous and has the functional value $\mu_A$ (x) at precisely one element. This element is called the vertex.
Usually we will be using triangular, or triangular shaped fuzzy numbers. A **triangular fuzzy number** is defined by three numbers m < n < p, where the base of the triangle is on the interval [m, p] and the vertex is at x = n. We write $\bar{N} = (m/n/p)$ for triangular Fuzzy number $\bar{N}$. A triangular shaped fuzzy number, written $\bar{N} \approx (m/n/p)$, has curves for its side instead of straight-line segments.

## Alpha-cuts

Let $\bar{N}$ be a fuzzy number. For $0 \le \alpha \le 1$, the alpha- cut of $\bar{N}$ written as $\bar{N}[\alpha]$,is defined as $\left\{ x \mid \bar{N}(x) \ge \alpha \right\}$

$\bar{N}[0]$ is defined as the closure of the union of $\bar{N}[\alpha]$ for $\alpha \in (0,1]$

### 7.3.3 Simulation of Fuzzy Continuous Systems

In order to solve fuzzy differential equations, one could only fuzzify the initial values, because the fuzzy solution became too difficult to obtain when more parameters became fuzzy. Instead, we can apply our knowledge about continuous simulation. Then, we can fuzzify more parameters. For instance, in the predator-prey model we may use fuzzy numbers $\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{x}_0, \bar{y}_0$.

### Types of Fuzzy Estimators

We will consider only two methods of fuzzy estimators:

1. expert opinion
2. from data using confidence intervals.

### Fuzzy Estimators Based on Expert Opinion

We want to estimate the value for a certain parameter b. First, assume that we have only one expert. Let $b_1$ be the smallest possible value for b, let $b_3$ be the largest possible value for b, and let $b_2$ be the most likely value. We can ask the expert to give values for $b_1$, $b_2$,

$b_3$, and we construct the triangular fuzzy estimator $\bar{b} = (b_1, b_2, b_3)$ for b. Now suppose we have N experts. We still want to construct a triangular fuzzy estimator $\bar{b} = (b_1 / b_2 / b_3)$. The easiest way to do this is to ask the experts for their $b_{1i}$, $b_{2i}$, $b_{3i}$ for all $1 \le i \le N$, and then take average of each component.

**Fuzzy Estimators Based on Data**

Let X be a random variable with probability density function f(x; θ) for single parameter θ . Assume that θ is unknown, and must be estimated from a random sample $X_1, \ldots X_n$. From statistics, we can learn to construct point estimation $\theta^*$ and (1 - β) 100% confidence interval θ. B is usually set to 0.10,0.05, or 0.01.
The trick is to find all (1 - β) 100% confidence intervals for $0.01 \le \beta \le 1$(starting at 0.01 is arbitrary). To this we add the interval $[\theta_*, \theta_*]$ for the 0% confidence interval. Now we place these intervals on top of each other, to produce a triangular shaped fuzzy number $\bar{\theta}$ whose α – cuts are the confidence intervals. To make it a complete fuzzy numbers, we will drop the graph of $\bar{\theta}$ straight down (Figure 2).

**Example of a Data-Based Fuzzy Estimator**

Consider $X \sim N(\mu,\sigma^2)$,with σ known and μ unknown. Suppose the mean of a random sample from $N(\mu,\sigma^2)$ turns out to be $\bar{x}$. We know that $\bar{x} \sim N(\mu, \sigma^2 / n)$, so $(\bar{x} - \mu)/(\sigma / \sqrt{n}) \sim N(0,1)$. So $P\left(-z_{\frac{\beta}{2}} \le \frac{\bar{x} - \mu}{\sigma / \sqrt{n}} \le z_{\frac{\beta}{2}}\right) = 1 - \beta$.
This leads to the (1 - β) 100% confidence intervals for
$\mu: [\theta_1(\beta), \theta_2(\beta)] = \left[\bar{x} - z_{\beta/2}\sigma / \sqrt{n}, \bar{x} + z_{\beta/2}\sigma / \sqrt{n}\right]$

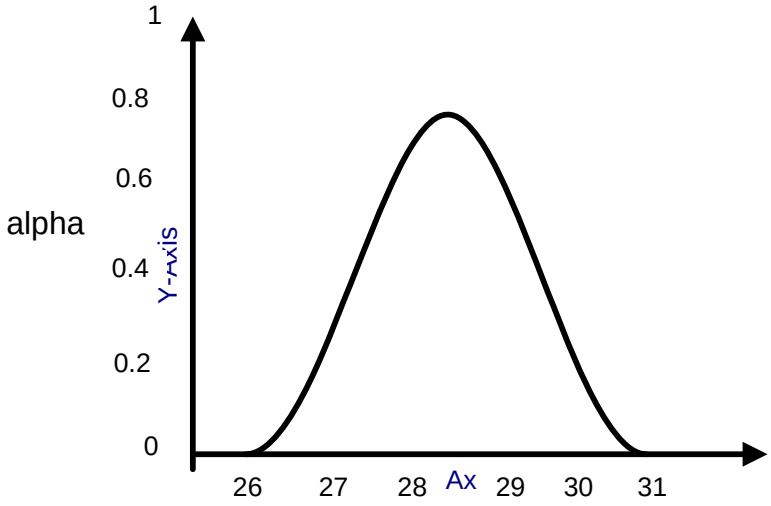



**Figure 2:Data Based Fuzzy Estimation**

## Example 1: Bungee Jumping Model

A bungee jumper jumps from 240 ft above the ground. The length of the unstretched bungee cord is 90 feet. The differential equation of motion is

$$m\frac{d^2x}{dt^2} = mg - F(x) - R(v)$$

where F(x) = kx is the force on the jumper exerted by the bungee cord, and R(v) = cv is air resistance with velocity v. Notice that $v = x_0(t)$.
A cord we plan to use has k = 2.5 pounds per foot. Is this cord good enough?

## 7.4 Summary

Database is very important in modeling and simulation ,when developed model is used large number of inputs data and produce large number of output.
Artificial Intelligence (AI) is to play an important role in the development of complex system simulation where decision support system is required. In addition, AI formalisms support the integration of knowledge from multiple sources at different levels of granularity, which could play an important role. Not only A.I. is used its newly born sub-fields like neural network and fuzzy sets are also used in modelling and simulation of very complex systems.

## 7.5 Key words

Database, Information Model, Simulation, Extensible Markup Language (XML), Artificial Intelligence, Computer Simulation, Genetic Algorithm, Cognitive Science, Neural Network, Fuzzy Sets, Agent, Crisp Set, Alpha Cuts.

## 7.6 Self-Assessment Questions

Q.1 Why we used database in Modelling and Simulation?. Explain with suitable example.

Q.2 Discuss the role of Artificial Intelligence in Modelling and Simulation.

Q.3 How Neural Network is applied in Modelling and Simulation of a complex system.

Q.4 How Fuzzy Set is applied in Modelling and Simulation of a complex system.

Q.5 Simulate the Bungee Jumping Model in a suitable language and analysis the results.

## 7.7 References/ Suggested Readings


1. Gregory L. Fenves, Frank McKenna Axel rod, R. (1997) ‘Data Model for Simulation
2. N. Deo , “ System Simulation”, Prentice Hall of India