Simulation is used extensively to simulate computer systems, because of their great importance to the everyday operations of business, industry, government and universities. The motivations for simulating computer systems, the different types of approaches used, and the interplay between characteristics of the model and implementation strategies are discussed below.

SIMULATION TOOLS

HISTORY OF SIMULATION SOFTWARE

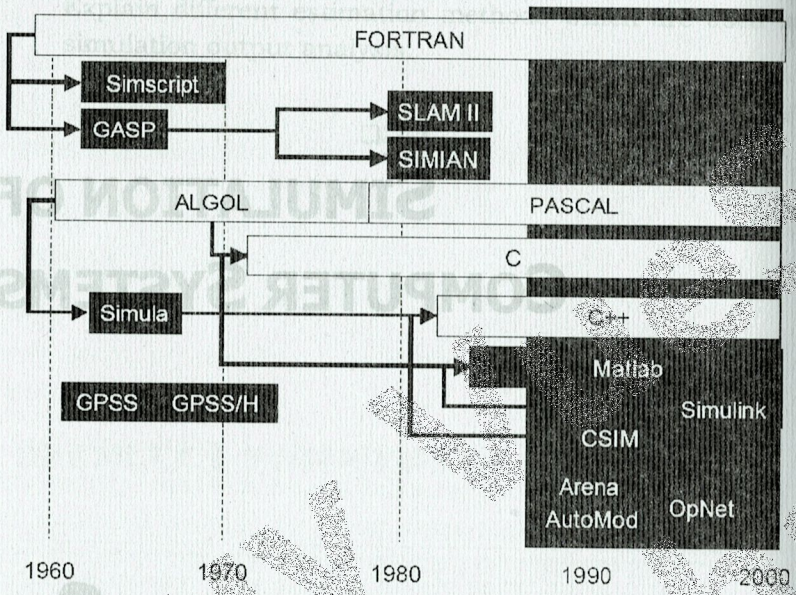


Figure 8.1: History of simulation tools

1. 1995 - 60 The Period of Search
   - Search for unifying concepts and the development of reusable routines to facilitate simulation.
   - Mostly conducted in FORTRAN
2. 1961 - 75 The Advent
   - Appearance of the forerunners of simulation programming languages (SPLs.)
   - The first process interaction SPL, GPSS was developed at IBM

3. 1966 - 70 The Formative Period
   - Concepts were reviewed and refined to promote a more consistent representation of each language's worldview
4. 1971 - 78 The Expansion Period
   - Major advances in GPSS came from outside IBM
   - GPSS/NORDEN, a pioneering effort that offered an interactive, visual online environment (in Norden Systems.)
   - GASP added support for the activity-scanning worldview and event scheduling worldview (at Purdue.)
5. 1979 - 86 The Period of Consolidation and Regeneration
   - Beginnings of PSLs written for, or adapted to, desktop computers and microcomputers.
   - Two major descendants of GASP appeared: SLAM II and SIMAN (provide multiple modeling perspectives and combined modeling capabilities).
6. 1987 – Now The Period of Integrated Environments
   - Growth of SPLs on the personal computer and the emergence of simulation environments with graphical user interfaces, animation and other visualization tools.
   - Recent advancements have been made in web-based simulation.

SELECTION OF SIMULATION SOFTWARE

- Advice when evaluating and selecting simulation software:
- Consider the accuracy and level of detail obtainable, ease of learning, vendor support, and applicability to your applications.
- Execution speed is important.
- Beware of advertising claims and demonstrations.
- Ask the vendor to solve a small version of your problem.

- Model building feature
- Runtime environment
- Animation of layout features
- Output features
- Vendor support and product documentation

Simulation Languages

As we have learned 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 discrete simulation language can be classified in several ways.

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.

CLASSIFICATION OF CONTINUOUS SIMULATION LANGUAGES

Continuous simulation language can be classified in two ways, as

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.

CONTINUOUS SYSTEM MODELING PROGRAM (CSMP):

CSMP III (Continuous System Modeling Program) is a computer program which is specifically designed to solve differential equations with a minimum of effort on the part of the user . The user need not know fancy numerical methods, nor must he know detailed computer programming . CSMP III represents the most recent phase in the evolution of digital programs for solution of differential equations. It is the most advanced simulation language program in widespread use today. Figure 8.1 shows the evolution of continuous system simulation languages (CSSL). CSMP/360 is only slightly different from CSMP III and most of the descriptions here will also apply to CSMP/360.

BASIC STRUCTURE

CSMP III is composed of three basic segments . The segments are divided as to function, type, and time of execution. Some segments in CSMP III are called procedural and others are called parallel . A procedural segment is executed sequentially ; the statements are executed in the order of appearance. A parallel segment is sorted by a phase of the CSMP III program into the proper sequence of execution . The following coding will best illustrate the differences in parallel and procedural sections.

```
B = A/2.
A = 10.0
C=2.*A
```

If the section were procedural the results would be A = 10. B = ? (unidentified since A was undefined when B was calculated)

```
C = 20.
```

If the section were a parallel section, the sequence of statements would be rearranged by a section of the CSMP III program to the following:

```
A = 10.0          A = 10.0
B = A/2.    or    C = 2.*A
C=2.*A            B=A/2.
```

The results would be :

```
A= 10
B= 5
C= 20
```

There are three basic segments of the CSMP III structure. They are INITIAL, DYNAMIC, and TERMINAL. The INITIAL segment is

a parallel segment which is executed at the beginning of the simulation . The TERMINAL section is a procedural section which is executed at the end of the simulation . The DYNAMIC section is the heart of the program where integration takes place . At this point an example will best illustrate the three CSMP III program sections.

A CSMP III program is constructed from three general types of statements:

1. Structural statements, which defined the model. They consist of FORTRAN-like statements, and functional blocks designed for operations that frequently occur in a model definition.
2. Data statements, which assign numerical values to parameters, constants, and initial conditions.
3. Control Statements, which specify options in the assembly and execution of the program, and the choice of output.

Structural statements can make use of the operations of addition, subtraction, multiplication, division, and exponentiation, using the same notation and rules as are used in FORTRAN, if, for example, the model includes the equation.

$$X = \frac{6Y}{W} + (Z - 2)^2$$

the following statement would be used:

$$X = 6.0 * \frac{Y}{W} + (Z - 2.0)**2.0$$

Note that real constants are specified in decimal notational. Exponent notation may also be used; for example, 1.2 E - 4 represents 0.00012. Fixed value constants may also be declared. Variable names may have up to six characters.

There are many functional blocks which, in addition to providing operations FORTRAN. Among these are the exponential functions, trigonometric functions, and function for taking maximum and minimum values. Figures 8.2 shows a list of eleven of the functional blocks.

The following functional block is used for integration:

```
Y = INTGRL (IC, X)
```

Where Y and X are the symbolic name of two variables and IC is a constant. The variable Y is the integral with respect to time of X, and it takes the initial value IC.

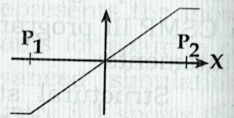
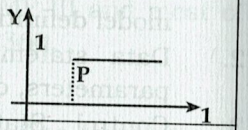
General Form	Function
Y = INTGRL (IC, X) Y (0) = IC INTEGRATION	$Y = \int_{t}^{0} xdt + IC$
Y = Limit (P_1, P_2, x) LIMITER	$y = P_1$ $x < P_1$ $y = P_2$ $x > P_2$ $Y = x$ $P_1 \le x \le P_2$ 
Y = STEP (P) STEP FUNCTION	$Y = 0$ $t < P$ $Y = 1$ $t \ge P$ 
Y =- EXP (X) EXPONENTIAL	$Y = e^x$
Y = ALOG (X) NATURAL LOGARITHM	Y = In (X)
Y = SIN (X) TRIGONOMETRIC SINE	Y = Sin (X)
Y = SQRT (X) SQUARE ROOT	$Y\ X^{1/2}$
Y = ABS (X) ABSOLUTE VALUE (REAL ARGUMENT AND OUTPUT)	Y = | X |
Y = AMAX1 (x_1, x_2, . . . x_n) Largest value (REAL ARGUMENTS AND OUTPUT)	y = max (X_1, X_2, . . . , X_n)
Y = AMIN1 (x_1, x_2, . . . x_n) SMALLEST VALUE (REAL ARGUMENTS AND OUTPUT)	y = max (X_1, X_2, . . . , X_n)

Figure 8.2: Functional blocks of Structural statements used in CSMP

Of the data statements, one called INCON can be used to set the initial values of the integration-function block. Other parameters can be given a value for a specific run with the CONST control statement. A control statement called PARAM can also be used to assign values to individual parameters, but its chief purpose is to specify a series of values for one parameter (and only one). The model will be run with the specified parameter taking each of the values on successive runs of the same model. Example of how these statements are written are:

CONST A = 0.5, XDOT = 1.25, YDOT = 6.22

PARAM D = (0.25, 0.50, 0.75, 1.0)

An indicated, several values can be specified with each statement by separating the values with commas.

Among the control statements is one called TIMER, which must be present to specify certain time intervals. An integration interval size must be specified. For adequate accuracy, it should be small in relation to the rate at which variables change value. The total simulation time must also be given. Output can be in the form of printed table and /or print-plotted graph s. Interval sizes for printing and plotting results need to be specified. The following is an example:

TIMER DELT = 0.005, FINTIM = 1.5, PRDEL = 0.1, OUTDEL = 0.1

The items specified are

DELT = Integration interval

FINTIM = Finish time

PRDEL = Interval at which to print results

OUTDEL = Interval at which to print-plot

If printed and/or print-plotted output is required, control statement with the words PRINT and PRTPLT are used, followed by the names of the variables to form the output. Several can be part of the output and they are listed with their names separated by commas. Two other control statements with the words TITLE and LABEL can be used to put headings on the printed and print plotted outputs, respectively. Whatever comment is written after the works becomes the heading.

The set of structural, data, and control statements for a problem can be assembled in any order but the must end the an END control statement. However control statements which define another run of the same model an follow and END statement if they also are terminated by another END statement. This can be repeated many times, unit an ENDJOB statement signals end of all runs. A completely separate model can then follow the ENDJOB statement.

As an example, Fig. 8.3 shows a CSMP III program for the automobile wheel suspension problem. It has been coded for the case where M = 2.0, F = 1, and K = 400, as specified in the CONST statement. A PARAM statement has been used to produced runs with different values of D, to give different values to the damping ratio. the damping ratio values are 0.1, 0.3, 0.7, 1.0 and 2.0. The program integrated with an interval of 0.005, and runs for a time of 1.5.

```
TITLE AUTOMOBILE SUSPENSION SYSTEM
*
PARAM d = (5.656, 16.968, 39.592, 56.56, 113.12)
*
x2dot = (1.0/m) * (k * F – K * X – D * XDOT)
XDOT  = INTGRL (0.0, X2DOT)
X = INTGRL (0, 0, XDOT)
*
CONST m = 2.0, F = 1.0, K = 400.0
TIMER DELT = 0.005, FINTIM = 1.5, PRDEL = 0.05, OUTDEL
T = 0.05
PRINT X, XDOT, X2DOT
PRTPLT X
LABEL DISPLACEMENT VERSUS TIME
END
STOP
```

Figure: 8.3. CSMP III CODING FOR THE AUTOMIBILE SUSPENSION PROBLEM

For example a complete CSMP program is used for simulation of Auto Pilot system.

Simulation of an Autopilot

In order to simulate the action of the autopilot, we first construct a mathematical model of the aircraft system. The error single, $\in$, has been defined as the difference between the desired heading, or input, θ_i, and the actual heading, or output, θ_0. We therefore have the following identity.

$$\in = \theta_1 - \theta_0$$

We assume the rudder is turned to an angle proportional to the error signal, so that the force changing the aircraft heading is proportional to the error signal.

Instead of moving the aircraft sideways, force applies a torques which will turn the aircraft. The strength of the torque, or turning force, depends on how far book the rudder is placed. However, just as the automobile movement was resisted by a shock absorber, the turning of the aircraft produces a resisting, viscous drag, approximately proportional to the angular velocity of the aircraft. The torque acting on the aircraft can, therefore, be represented by the following equation.

$$\text{Torque} = K\in - D\dot{\theta}$$

Where K and D are constants. The first term on the right-hand side is the torque produced by the rudder, and the second is the viscous drag.

When discussing differential equation in unit one, it was stated that the fundamental law of mechanics is that the acceleration of a body is proportional to the applied force. That was related to linear motion. The same law, however, applies to a turning motion: the angular acceleration of a body is proportional to the applied torque. Further, the coefficient of proportionality is the inertia of the body, denoted by I. Since the angular acceleration of the aircraft is the second derivative of its heading, the equation of motion is

$$I\ddot{\theta}_0 = \text{torque}$$

Substituting and transposing terms for mechanical system equation as mentioned in unit one, , the resultant equation is

$$I\ddot{\theta}_0 + D\dot{\theta}_0 + K\theta_0 = K\theta_t$$

If we divided both side of the equation by I, and make the following substitutions

$$2\zeta\omega = \frac{D}{I}, \omega^2 = \frac{K}{T},$$

The equation of motion relating output to input then takes the following form:

$$\ddot{\theta}_0 + 2\zeta\omega^2\theta_0 = \omega^2\theta_i$$

This is a second-order differential equation. Referring back to unit one, it will be seen that it is in exactly the same form as of dynamic mathematical model, which describes the relationship between output and input for the automobile wheel suspension system.

Suppose the aircraft is initially flying a stead course which, by definition, we take to be the zero heading. If it is asked to change to a new heading at time zero, this corresponds to a unit step change of input. Because of the correspondence between the equations of motion we have just noted, the results will be the same as shown in Fig., which showed the response of the suspension system to a unit step function. The results show that the aircraft will settle to the new heading, and the response will be oscillatory if ζ is less than 1.

To simulate how the autopilot can be designed to modify the aircraft response, it is more convenient to leave the model in the form of the three individual equations: Eqs. (8.1), (8.2), (8.3). Using the variable ERROR and TORQUE to represent the error signal and the applied torque, the equations are

ERROR = θi – Q0

TORQUE = K★ERROR – D ★θ0

Iθ̈ 0 = TORQUE

If we also use the variable HEAD, ANGVEL, and ANGACC to represent the aircraft heading and its first two derivatives, respectively, together with INPUT for the desired heading, the equations can be written

ERROR = INPUT – HEAD

TORQUE = K★ERROR – D★ANGVEL

I★ANGACC = TORQUE

A CSMP III program for the system is shown in Fig. 8.4. Instead of using a step function change of heading, which simple repeat the automobile wheel case, the aircraft is being asked to turn in a circle. The desired heading is then continually increasing at a uniform rate so that

INPTU = A★TIME

Where A is a constant and TIME is a CSMP variable representing the time, t. The constants K, I, and A have been set to values of 400, 2.00, and 0.0175, respectively. The constant D has been programmed to take different values of five separate runs, so that the damping ratio, ζ, will have the values 0.1, 0.3, 0.7, 1.0 and 2.0.

```
TITLE ALRCRAFT WITH RATE CONTROL
*
PARAM D = (5.656, 16.968, 39.592, 56.56, 113.12)
INPUT = A* TIME
ERROR = INPUT
TORQUE = K*ERROR – D*ANGVEL
ANGACC = TORQUE/I
HEAD = INTGRL (0, 0, ANGACC)
*
CONST I = 2.0,K = 400.0, A = 0.0175
TIMER DELT = 0.005, FINTIM = 1.5, PRDEL = 0.05
PRINT HEAD
LABEL HEADING VERSUS TIME
END
```

Figure 8.4: CSMP III program for aircraft with rate control.

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.

Classification of Discrete Simulation Languages

Discrete simulation language can be classified in four ways

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.

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 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:
   - Parameters
   - Attributes
   - Methods
   - Life, that represents the activity started upon object creation.

   Objects can interact in these ways:
   - Direct access to parameters and attributes
   - Mutual calling of methods
   - 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.

GENERAL PURPOSE SIMULATION SYSTEM

GPSS PROGRAMS

The General Purpose simulation system language has been developed over many years, principally by the IBM Corporation. Originally published in 1961, it has evolved through several versions. It has been implemented on several different manufacturers' machines, and there are variations in the different implementations. With regard to the successive IBM versions of the language, all but GPSS/360 and GPSS V, are obsolete. Of the two current versions, GPSS/360 models can operate with GPSS V, with some minor exceptions and modification. GPSS V, however, is more powerful and has more language statements and facilities. If these extensions are avoided. GPSS V models will run under GPSS/360.

The description here will be of GPSS V as implemented by the IBM Corporation(8). Some of the more significant differences between GPSS/360 and GPSS V will be noted. For simplicity, the language will be called GPSS except where comparisons are being made.

In earlier versions of the program, GPSS stood for General Purpose System Simulation.

GENERAL DESCRIPTION

The system to be simulated in GPSS is described as a block diagram in which the blocks represent the activities, and lines joining the blocks indicate the sequence in which the activities can be executed. Where

there is a choice of activities more than one line leaves a block and the condition for the choices is stated at the block.

The use of block diagrams to describe systems is, of course, very familiar. However, the form taken by a block diagram description usually depends upon the person drawing the block diagram description usually depends upon the person drawing the block diagram. To base a programming language on this descriptive method, each block must be given a precise meaning. The approach taken in GPSS is to define a set of 48 specific block types, each of which represents a characteristics action of systems. The program use must draw a block diagram of the system using only these block types. Each block type is given a name that is descriptive of the block action and is represented by a particular symbol. Figure 8.4 shows the symbols used for the block types.

Generate Block

The GENERATE Blocks can be thought of as a door through which transactions enter a model. Transactions can be made to enter the model at different points in time. The time between two consecutive transaction arrivals at a given GENERATE block is called inter-arriva1 time. The block to which the transactions move is the next sequential block following the GENERATE block. There can be any number of generate blocks in a mode 1. When the GPSS program encounters a GENERATE block one transaction is immediately created. The set of four basic fields for the new transaction is obtained from the internal chain of inactive transactions, which is a stack.

Terminate Block

This block always accepts transactions and transactions are removed from a model whenever they move into a TERMINATE block. There may be any number of terminate blocks in the model. The A operand of the terminate block is called the termination counter. The termination counter is a special counter in which a positive integer value is stored at the time a simulation is begun. As the simulation proceeds, when the

transactions move into a terminate block with the A operand, this counter is decremented. When this counter reaches a value of zero or less, the simulation stops. Though there may be more than one termination block, there is only one termination counter and this termination counter will be decremented whenever a transaction moves into any terminate block in a model. The termination counter is supplied with its initial value at the time simulation begins. It uses the A operand of the START blocks as the initial value for the termination counter. Movement of transactions into the TERMINATE block which does not have the A operand does not decrease the value of the termination counter. START The GPSS processor starts the simulation when it encounters a START block. It uses the A operand on the START block as the initial value of the termination counter. The TERMINATE block and START block are used in harmony to control the duration of the simulation run.

SEIZE & RELEASE In GPSS the term "facility" is a synonym for "Server". Just as there can be many servers at different points in a system, there can be many facilities in a GPSS model. Names are given to facilities making it possible to distinguish among them. The names are given by the model builder. When the server is being used the following steps are followed:

1. The transaction waits for its turn, if needed.
2. When its turn comes , it engages the server.
3. It holds the server in a state of capture while the service demands are performed.
4. When the demanded service has been performed, the server is released .

Advance

The ADVANCE block accomplishes the task of freezing a transaction's motion for a prescribed length of time. The information required to describe the applicable service time distribution is expressed through the advance block's A and B operands.

QUEUE & DEPART The above blocks are to some extent similar to the Facility- Depart blocks. When a transaction moves into the QUEUE block, the event "join a waiting line " is simulated and when a transaction moves into the DEPART block the event "depart a waiting line " is simulated. Statistics describing key features like the following are gathered.

* Total entries to the waiting line.
* Total number of entries with zero waiting time.
* The maximum queue length.
* Average number waiting.
* Average waiting time.
* Current content of the queue.

On many occasions, even when waiting is known to occur at certain points in a model, the model builder chooses not to use the queue entity at those points, when he is not interested in gathering statistics at those points. The result of not using the QUEUE- DEPART blocks is a savings in computer time.

TRANSFER This block is used to divert the transactions to some non-sequential block is a GPSS model. It could be used in a conditional or unconditional or random transfer. Thus it could be used in more than one mode.

ENTER & LEAVE The parallel servers are represented by the above blocks. The transaction captures the parallel server. The transaction waits, if necessary, then it captures a server, it holds the server in a state of capture over some interval of time and finally it releases the server. Before making use of the ENTER and LEAVE blocks the statement STORAGE associated with it should be defined. This is the capacity definition statement.

PRIORITY When a transaction enters a model, its priority level is set equal to the value specified by the E operand at its GENERATE block. The priority level value influences the chronological sequence in which various transactions move forward in a model. The current transactions priority value can be changed by the PRIORITY block.

ASSIGN Each transaction in a GPSS model possesses a set of a maximum of 100 parameters. As a transaction moves through a model, its parameter values can be assigned and modified as the modeller wishes. The parameter "n" is referred to as Pn and it is not possible to name them symbolically. The parameters can only be assigned integer values and the meaning of the parameter is determined by the analyst. The initial values of all the parameters are set to zeros.

TEST The relation between the values of two standard numerical attributes can be examined by the use of the TEST block. The A and B operands are the names of the two standard numerical attributes involved. An auxiliary operator X, indicates the way the two attributes are to be compared against each other.

GATE The GATE block's A operand states the name of the logic switch to be tested. When the transaction finds the GATE closed, it is held at the block preceding the GATE, contributing to the current count there. CLEAR This clears all model statistics and sets than to zero. This also removes all the transactions in the model at the end of the simulation. This also re initializes the counter.

RESET In most cases, the initial model consolidations may vary markedly from those when a steady state is reached. In order for the simulation to be realistic the statistics collected during the initial phase have to be discarded so that the statistics corresponding to the steady stock of the model are available for analysis. This is achieved by use of the RESET block. A frequent practice is to use the model itself, in experimental fashion, to estimate the duration of simulation required to reach steady state. The RESET card can be used for this experimentation.

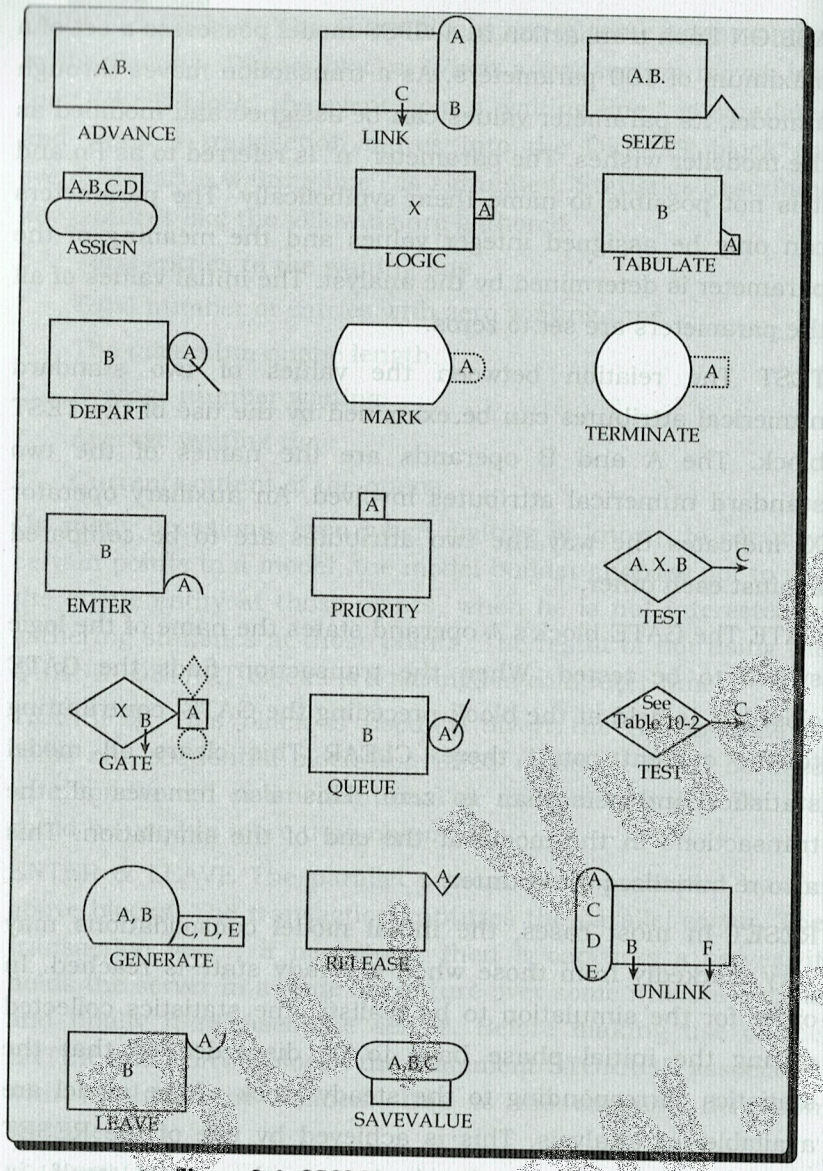


Figure 8.4: GPSS block-diagrams symbols

Coding instructions for all the described block types are shown in table 8.1 and table 8.2 describes the control statement. Each block type has a number of data fields. As the blocks are described, the fields will be referred to as field A, B, C, and so on, reflecting the order in which they are specified.

Operation	A	B	C	D	E	F
ADVANCE	Mean	Modifier	(Funct. No.)	Param. type		
ASSIGN	Param. NO. (±)	Source				
DEPART	Queue No.	(Units)				
ENTER	Storage No.	(Units)				
Gate†	Item No.	(Next block B)				
GENERATE	Mean	Modifier	(Offset)	(Count)	(Priority)	(Params.)
LEAVE	Storage No.	(Units)				
LINK	Chain No.	Order	(Next block B)			
LOGIC {R S I}	Switch No.					
MARK	(Param. No.)					
PRIORITY	Priority					
QUEUE	Queue No.	(Units)				
RELEASE	Facility No.					
SAVEVALUE	S.V. No. (±)	SNA				
SEIZE	Facility No.					
TABULATE	Table No.	(Units)				
TERMINATE	(Units)					
TEST†	Arg. 1	Arg. 2	(Next block B)			
TRANSFER	Select. Factor	Next block A	Next block B			
UNLIKE	Chain No.	Next block A	Count	(Param. No.)	(Arg.)	(Next block B)

Location	Operation	A	B	C	D
	CLEAR				
	END				
Function No.	FUNCTION	Argument	{C, D, L} No. of Points		
	INITIAL	Enitity	Value		
	JOB				
	RESET				
	SIMULATE				
	START	Run Count	(NP)		
Storage No.	STORAGE	Capacity			
Table No.	TABLE	Argument	Lower limit	Interval	No. of Intervals

Moving through the system being simulated are entities that depends upon the nature of the system. For examples a communication system is concerned with the movement of message, a road transportation system with motor vehicles, a data processing system with records, and so on. In the simulation, these entities are called transaction. The sequence of events in real time is reflected in the movement of transactions from block to block in simulated time.

Transactions are crated at one or more GENERATE blocks and are removed from the simulation at TERMINATE blocks. There can be many transactions simultaneously moving through the block diagram. Each transactions is always positions at a block and most blocks diagram. Each transaction is always positioned at a block and most blocks can hold many transactions simultaneously. The transfer of a transaction from one block to another occurs instantaneously at a specific time or when some change of system condition occurs.

A GPSS block diagram can consist of many blocks up to some limit prescribed by the program (usually set to 1,000). An identification number called a location is given to each block, and the movement of transactions is usually from one block to the block with the next highest location. The locations are assigned automatically by an assembly program within GPSS so that, when a problem is coded, the blocks are listed in sequential order. Blocks that need to be identified in the programming or problems (for example, as points to which a transfer is to be made) are given a symbolic name. The assembly program will associated the name with the a appropriate location.

Symbolic name of blocks and other entities of the program must be from three to five on-blank characters of which the first three must be letters.

For example: Simulation of a Manufacturing Shop

To illustrate the features of the program described so far, consider the following simple example. A machine tool in a manufacturing shop is turning out parts at the rate of one every 5 minutes. As they are finished, the parts go to an inspector, who takes 4 ± 3 minutes to examine each one and rejects about 10% of the parts. Each part will be represented by one transaction, and the time unit selected for the problem will be 1 minute.

A block diagram representing the system in fig. 8.5.

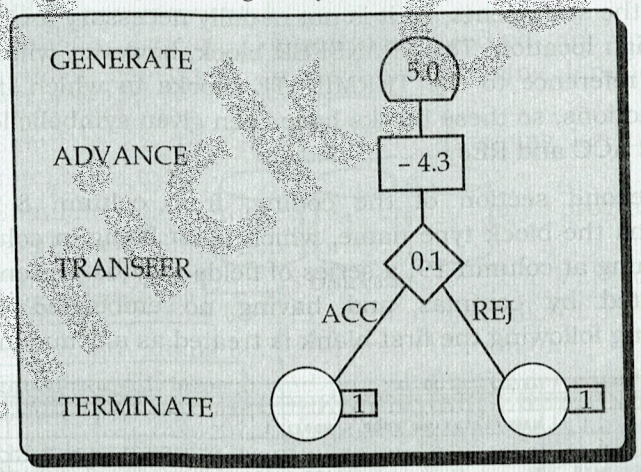


Figure 8.5: Manufacturing shop-model 1

The usual convention used in drawing block is to place the block location (where needed) at the top of the block; the action time is indicated in the center in the form T = a, b where a is the mean and b is the modifier; and the selection factor is placed at the bottom of the block.

A GENERATE block is used to represent the output of the machine by creating one transaction every five units of time. An ADVANCE block with a mean of 4 and therefore of 3 is used to represent inspection. The time spent on inspection will therefore be any one of the values 1, 2, 3, 4, 5, 6, or 7 with equal probability given to each value. Upon completion of the

inspection, transaction go to a TRANSFER block with a selection factor of 0.1, so that 90% of the parts go to the next location (exit 1) called ACC, to represent accepted parts and 10% go to another location (exit 2) called REJ to represent rejects. Since there is not further interest in following the history of the parts in this simulation, both locations reached from the TRANSFER block are TERMINATE blocks.

The problem can be coded in a fixed format as shown below. Column 1 is only used for a comment card. An * in column 1 results in the statement being printed in the output only. A field from columns 2 to 6 contains the location of the block where it is necessary for it to be specified. The GPSS program will automatically assign sequential location numbers as it reads the statements, so it is not usually necessary for the user to assign location. The TRANSFER block, however, will need to make reference to the TERMINATE blocks to which it sends transactions, so these blocks have been given symbolic location names ACC and REJ.

The second section of the coding, from column 8 to 18, contains the block type name, which must being in column 8. Beginning at column 19, a series of fields may be present, each separated by commas and having no embedded blanks. Anything following the first blank is treated as a comment.

LOCATION	OPERATION	A,B,C,D,E,F	
*		MANUFACTURING SHOP - MODEL 1	
*			
	GENERATE	5	CREATE PARTS
	ADVANCE	4,3	INSPECT
	TRANSFER	.1,ACC,REJ	SELECT REJECTS
ACC	TERMINATE	1	ACCEPTED PARTS
REJ	TERMINATE	1	REJECTED PARTS
	START	1000	

SIMULATION OF COMPUTER SYSTEM

Computer systems are incredibly complex. A computer system exhibits complicated behavior at time scales from the time it takes to "flip" a transistor's state (on the order of 10^{-11} seconds) to the time it takes a human to interact with it (on the order of seconds or minutes). Computer systems are designed hierarchically, in an effort to manage this complexity. Figure 8.6 below illustrates the point.

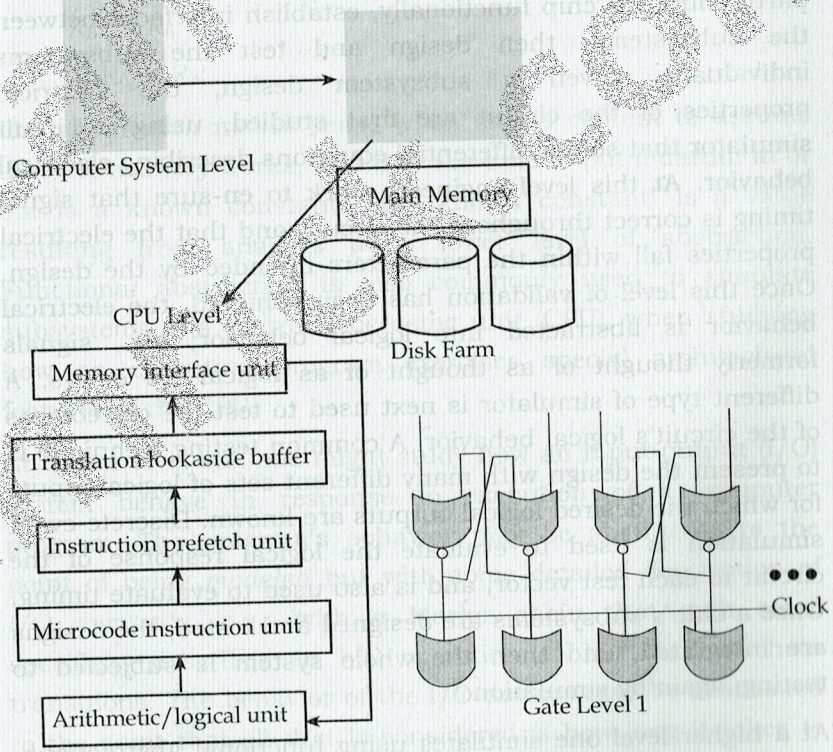


Figure 8.6: Simulation of computer system

DIFFERENT LEVELS OF ABSTRACTION

- At a high level of abstraction (the system level) one might view computational activity in terms of tasks circulating among servers, queueing for service when a server is busy.

- A lower level in the hierarchy can view the activity among components of a given processor (its registers, its memory hierarchy).
- A lower level still one views the activity of functional units that together make up a central processing unit.
- At an even lower level one can view the logical circuitry that makes it all happen.

Simulation is used extensively at every level of this hierarchy, with results from one level being used at another. For instance, engineers working on designing a new chip will begin by partitioning the chip functionally, establish interfaces between the subsystems, then design and test the subsystems individually. Given a subsystem design, the electrical properties of the circuit are first studied, using a circuit simulator that solves differential equations describing electrical behavior. At this level engineers work to en-sure that signal timing is correct throughout the circuit, and that the electrical properties fall within the parameters intended by the design. Once this level of validation has been achieved, the electrical behavior is abstracted into logical behavior; e.g., signals formerly thought of as thought of as logical 1's or 0's. A different type of simulator is next used to test the correctness of the circuit's logical behavior. A common testing technique is to present the design with many different sets of logical inputs for which the desired logical outputs are known. Discrete-event simulation is used to evaluate the logical response of the circuit to each test vector, and is also used to evaluate timing. Once a chip's subsystems are designed and tested, the designs are integrated, and then the whole system is subjected to testing, again by simulation.

At a higher level one simulates using functional abstractions. For instance, a memory chip could be modeled simply as an array of numbers and a reference to memory as just an indexing operation. A special type of description language exists for this level, called "register-transfer language". This is like a programming language, with pre assigned names for

registers and other hardware-specific entities, and with assignment statements used to indicate data transfer between hardware entities. For example, the sequence below loads into register R_3 the data whose memory address is in register R6, subtracts one from it, and writes the result into the memory location that is word-adjacent (a word in this example is 4 bytes in size) to the location first read.

$R_3 = M[R_6];$

$R_3 = R_3 - 1;$

$R_6 = R_6 + 4;$

$M[R_6] = R_3;$

The abstraction makes sense when one is content to assume that the memory works, and that the time to put a datum in or out is a known constant. The "known constant" is a value resulting from analysis at a lower level of abstraction. Functional abstraction is also commonly used to simulate subsystems of a central processing unit (CPU), when studying how an executing program exercises special architectural features of the CPU.

At higher level still one might study how an input-output (I/O) system behave in response to execution of a computer program. The program's behavior may be abstracted to the point of being *modeled* but with some detailed description of I/O demands, e.g., with a Markov-chain that with some specificity describes an I/O operation as the Markov chain transitions. The behavior of the I/O devices may be abstracted to the point that all that is considered is how long it takes to complete a specified I/O operation. Because of these abstractions one can simulate larger systems, and simulate them more quickly. The execution of a program is modeled with a randomly sampled CPU service interval; its I/O demand is modeled as a randomly sampled service time on a randomly sampled I/O device.

MODEL INPUT

There are different means of providing input to a model. The model might be driven by

- Stochastically generated input, using either simple or complicated operation assumptions, used at the high end of the abstraction. Useful for the study he system behavior over a range of scenarios.
- Trace input, measured from actual systems used at lower levels.

High-level systems simulations accept a stream of job descriptions; CPU simulations accept a stream of instruction descriptions, memory simulations accept a stream of memory references; and gate-level simulations accept a stream of logical signals.

Computer systems modeled as queueing networks typically interpret "customers" as computer programs; servers typically represent services such as attention by the CPU or an input/output system.

Random sampling generates customer inter arrival times, and it may also be used to govern routing and time-in-service.

However, it is common in computer systems contexts to have routing and service times be state dependent; e.g., the next server visited is already specified in the customer's description, or may be the attached server with least queue length.

MODULATED POISSON PROCESS

An input model that is sometimes used to retain a level of tractability is a modulated Poisson process, or MPP.

The underlying framework is a continuous-time Markov chain (CTMC). A CTMC is always in some *state;* for descriptive purposes states are named by the integers: 1, 2,....

The CTMC remains in a state for a random period of time, transitions randomly to another state, stays there for a random period of time, transitions again, and so on.

The CTMC behavior is completely determined by its *generator matrix, Q ={q,j}*. For states i<> *j*, entry q,j describes the rate at which the chain transitions from state i into state *j*.

The rate describes how quickly the transition is made; its units are transitions per unit simulation time.

Diagonal element *q,i* is the negated sum of all rates out of state *i : q,i* = - € $q_{i,j}$ An operational view of the CTMC is that upon entering a state *i* , it remains in

that state for an exponentially distributed period of time, the exponential having rate – q_{ii} Making the transition, it chooses state *j* with probability – $q_{i,j}/ q_{i,i}$.

Many CTMCs are *ergodic,* meaning that, left to run forever, every state is visited infinitely often.

In an ergodic chain Π_I, denotes state i's stationary *probability, which* we can interpret as the long-term average fraction of time the CTMC is in state i.

A critical relationship exists between stationary probabilities and trans t on rates: for every state i,

$\pi i \sum {}^{q}i,j^{=} \sum \pi j {}^{q}j,i$

VIRTUAL MEMORY REFERENCING

Andomness can also be used to drive models in the middle levels of abstraction. In such a system, the data and instructions used by the program are organized in units called *pages.* All pages are the same size, typically 2^{10} to 2^{12} bytes in size.

The physical memory of a computer is divided into *page frames,* each capable of holding exactly one page.

The decision of which page to map to which frame is made by the operating system.

As the program executes, it makes memory references to the "virtual memory," as though it occupied a very large memory starting at address 0, and is the only occupant of the memory.

On every memory reference made by the program, the hardware looks up the identity of the page frame containing the reference, and translates the virtual address into a physical address.

The hardware may discover that the referenced page is not present in the main memory; this is called *& page fault.*

When a page fault occurs, the hardware alerts the operating system, which then takes over to bring in the referenced page from a disk and decides which page frame should contain it.

The operating system may need to evict a page from a page frame to make room for the new one.

The policy the operating system uses to decide which page to evict is called the "replacement policy".

The quality of a replacement policy is often measured in terms of the fraction of references made whose page frames are found immediately, the *hit ratio.*

Virtual memory systems are used in computers that support concurrent execution of multiple programs.

In order to study different replacement policies one could simulate the memory-referencing behavior of several different programs, simulate the replacement policy, and count the number of references that page-fault.

Virtual memory works well precisely because programs do tend to exhibit a certain type of behavior, so-called locality of reference.

The program references tend to cluster in time and space, that when a reference to a new page is made and the page is brought in from the disk, it is likely that the other data or instructions on the page will also soon be referenced.

In this way the overhead of bringing in the page is amortized over all the references made to that page before it is eventually evicted.

A program's referencing behavior can usually be separated into a sequence of "phases," where during each phase the program makes references of a relatively small collection of pages called its *working set.*

Phase transitions essentially change the program's working st. The challenge for the operating system is to recognize when the pages used by a program are no longer in its working set, for these are the pages it can safely evict to make room for pages that *are* in some program's working set.

HIGH-LEVEL COMPUTER SYSTEM SIMULATION

Here we illustrate concepts typical of high-level computer simulations, by sketching a simulation model of a computer system that services requests from the World Wide Web.

Example

- A company that provides a major web site for searching and links to sites for Travel, commerce, entertainment, and the like wishes to conduct a capacity planning study.
- The overall architecture of their system is shown in Figure.
- At the back end one finds data servers responsible for all aspects of handling specific queries and updating databases.
- Data servers receive requests for service from application servers— machines dedicated to running specific applications supported by the site.
- In front of applications are web servers that manage the interaction of applications with the World Wide Web, and the portal to the whole system is a load-balancing router that distributes requests directed to the web site among web servers.
- The goal of the study is to determine the site's ability to handle load at peak periods.
- The desired output is an empirical distribution of the access response time.
- Thus, the high-level simulation model should focus on the impact of timing at each level that is used, system factors that affect that timing, and the effects of timing on contention for resources.

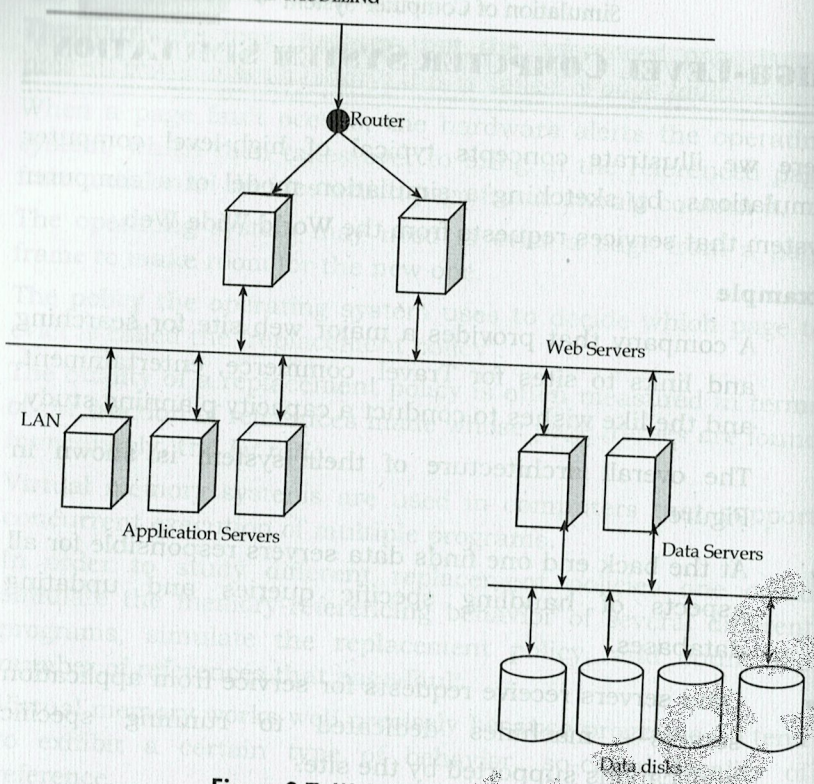


Figure 8.7: Website Server System

- To understand where those delays occur let us consider the processing associated with a typical query.
- All entries into the system are through a dedicated router, which examines the request and forwards it to some web server.
- It is reasonable to assume one switching time for a preexisting request and a different time for a new request.
- The result of the first step is selection of a web server, and enqueueing there of a request for service.
- A web server can be thought of as having a queue of threads of new requests, a queue of threads that are suspended awaiting a response from an application server, and a queue of threads "ready" to process responses from application servers.

- An accepted request from the router creates a new request thread.
- The servicing of a new request amounts to identification of an application and associated application server.
- A request for service is formatted and forwarded to an application server, and the requesting thread joins the suspended queue.
- At an application server, requests for service are organized along application types.
- A new request creates a thread that joins a new request queue associated with the identified application.
- An application request is modeled as a sequence of sets of requests from data servers, interspersed with computational bursts — e.g., burst 1 request data from D_1, D_3 and D_5 burst 2 request data from D_1 and D_2 burst D_3.
- An application server will implement a scheduling policy over se s of ready application threads.
- A data server creates a new thread to respond to a data request and places it in a queue of ready threads.
- A data server may implement memory-management policies, and may require further coordination with the application server to know when to release used memory.
- Upon receiving service the thread requests data from a disk, and suspends until the disk operation completes, at which point the thread is moved from the suspended list to the ready list, and when executed again reports back to the application server associated with the request.
- The thread suspended at the application server responds; eventually the application thread finishes and reports its completion back to the web-server thread that initiated it, which in turn communicates the results back over the Internet.

The web-site model is an excellent candidate for a threaded (process-oriented) approach to modeling. There are two natural approaches for defining a process. One emulates our CSIM

example, defining each request as a process. The model would be expressed from the point of view of a request going through the router, to a web server, and so on.

Disadvantage of query centered modeling

One complication with equating a query with a process is that it is not an exact fit to what happens in this model. A query passed from application server to data server may actually cause multiple concurrent requests to the data disks-it is insufficient to "push" a query process from router to disks and back. A query process can spawn concurrent supporting processes, which implies encoding a fork-and-join synchronization mechanism (a process spawns new processes, and waits for them to return). The CSIM example illustrated this by having the main thread suspend, once all job arrival processes had been generated, until the last one finished.

Advantage of query centered modeling

The advantage to a query-centered modeling approach is clarity of expression, and the ease with which query-oriented statistics can be gathered. A given query process can simply measure its delays at every step along the way, and upon departing the system include its observations in the statistical record the simulation maintains for the system.

An alternative process-oriented approach is to associate processes with servers. The simulation model is expressed from an abstracted point of view of the servers' operating system. Individual queries become messages that are passed between server processes. An advantage of this approach is that it explicitly exposes he scheduling of query processing at the user level. The modeler has both he oppor un ty and the responsibility to provide the logic of scheduling actions that mod l processing done on behalf of a query. It is a modeling viewpoint that simplifies analysis of server behavior — an overloaded server is easily identified by the length of its queue of runnable queries. However, it is a modeling viewpoint that is a bit lower in abstraction than the first one, and requires more modeling and coding on the part of the user.

An event-oriented model of this system need not look a great deal different than the second of our process-oriented models. Queries passed as messages between servers have an obvious event-oriented expression. A modeler would have to add to the logic, events, and event handlers that describe the way a CPU passes through simulation time. In an event-oriented model one would need to define events that reflect "starting" and "stopping" the processing of a query, with some scheduling logic interspersed. Additional events and handlers need to be defined for any "signaling" that might be done between servers in a process-oriented model. A process-oriented approach, even one focused on servers rather than queries, lifts model expression to a higher level of abstraction and reduces the amount of code that must be written. In a system as complex as the web site, one must factor complexity of expression into the overall model-development process.

CPU SIMULATION

The main challenge to making effective use of a CPU is to avoid stalling it, which happens whenever the CPU commits to executing an instruction whose inputs are not all present. A leading cause of stalls is the latency delay between CPU and main memory, which can be tens of CPU cycles.

One instruction may initiate a read, e.g., load $2, 4($3) which is an assembly-language statement that instructs the CPU to use the data in register 3 (after adding value 4 to it) as a memory address, and to put the data found at that address in register 2. If the CPU insisted on waiting for that data to appear in register 2 before further execution, the instruction could stall the CPU for a long time if the referenced address was not found in the cache. High-performance CPUs avoid this by recognize g that additional instructions can be executed, up to the point where the CPU attempts to execute an instruction that reads the contents of register 2, e.g., add $4, $2, $5.

This instruction adds the contents of registers 2 and 5 and places the result in register 4. If the data expected in register 2

is not yet present, the CPU will stall. To allow the CPU to continue past a memory load it is necessary to mark the target register as being unready allow the memory system to asynchronously load the target register while the CPU continues on in the instruction stream stall the CPU if it attempts to read a register marked as unready Clear the unready status when the memory operation completes.

The sort of arrangement described above was first used in the earliest supercomputers, designed in the 1960s. Modern microprocessors add some additional capabilities to exploit instruction-level parallelism (ILP).

PIPELINING

- The technique of pipelining has long been recognized as a way of accelerating the execution of computer instructions.
- Pipelining exploits the fact that each instruction goes sequentially through several stages in the course of being processed; separate hardware resources are dedicated to each stage, permitting multiple instructions to be in various stages of processing concurrently.
- A typical sequence of stages in an ILP CPU are as follows:

Instruction fetch: the instruction is fetched from the memory.

Instruction decode: the memory word holding the instruction is interpreted to determine what operation is specified; the registers involved are identified.

Instruction issue: an instruction is "issued" when there are no constraints holding it back from being executed. Constraints that keep an instruction from being issued include data not yet being ready in an input register, and unavailability of a functional unit (e.g., arithmetic-logical-unit) needed to execute the instruction.

Instruction execute: the instruction is performed.

Instruction complete: results of the instruction are stored in the destination register.

Instruction graduate : executed instructions are graduated in the order that they appear in the order that they appear in the instruction stream.

MEMORY SIMULATION

- One of the great challenges of computer architecture is finding ways o effectively deal with the increasing gap in operation speed between CPUs and main memory.
- The main technique that has evolved is to build hierarchies of memories.
- A relatively small memory—the L_1 cache—operates at CPU speed. A larger memory—the L_2 cache— is larger, and operates more slowly.
- The main memory is larger still, and slower still. The smaller memories hold data that was referenced recently.
- Data moves up the hierarchy on demand, and ages out as it becomes disused, to make room for the data in current use.
- For instance, when the CPU wishes to read memory location 100000, hardware will look for it in the L_1 cache, and failing to find it there, look for it in the L_2 cache.
- If found there, an entire block containing that reference is moved from the L_2 cache into the L_1 cache.
- If not found in the L_2 cache, a block of data containing location 10000 is copied from the main memory to the L_2 cache, and part of that block is copied into the L_1 cache.
- It may take 50 cycles or more to accomplish this.
- After this cost has been suffered, the hope and expectation is that the CPU will continue to make references to data in the block brought in, because accesses to L_1 data are made at CPU speeds.
- After a block ceases to be referenced for a time, it is ejected from the L_1 cache.

- It may remain in the L_2 cache for a while, and be brought back into the L_1 cache if any element of the block is referenced again.
- Eventually a block remains unreferenced long enough so that it is ejected also from the L_2 cache.

An alternative method copies back a block from one memory level to the lower level, at the point the block is being ejected from the faster level. The write-through strategy avoids writing back blocks when they are ejected, whereas the write-back strategy requires that an entire block be written back when ejected, even if only one word of the block was modified, once.

The principal measure of the quality of a memory hierarchy is its hit ratio at each level. As with CPU models, to evaluate a memory hierarchy design one must study the design in response to a very long string of memory references. Direct-execution simulation can provide such reference stream, as can long traces measured reference traffic. Nearly every caching system is a demand system, which means that a new block is not brought into a cache before a reference is made to a word in that block.

Set Associativity

- The concept of set associativity arises in response to the cost of the mechanism used to look for a match.
- Imagine we have an L_2 cache with 2 million memory words.
- The CPU references location 10000—the main memory has, say, 2 so the L_2 cache holds but a minute fraction of the main memory.
- How does the hardware determine whether location 10000 is in the L_2 cache? It uses what is called an associative memory, one that associates search keys with data.
- One queries an associative memory by providing some search key. If the key is found in the memory, then the data associated with the key is returned, otherwise indication of failure is given.

Fully Associative

- A fully associative cache is one where any address can appear anywhere in the cache.
- The idea is to partition the address space into sets. Figure given illustrates how a 48-bit memory address might be partitioned in key, set id, and block offset.

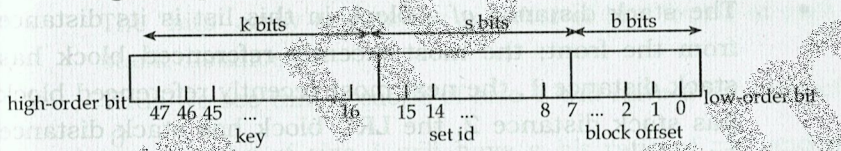


Figure 8.8: 48-bit address partitioned for cache

- Any given memory address is mapped to the set identified by its set id address bits.
- This scheme assigns the first block of 2^b addresses to set 0, the second block of 2^b addresses to set 1, and so on, wrapping around back to set 0 after 2^s blocks have been assigned.
- Each set is given a small portion of the cache—the set size—which typically is 2 or 4 or 8 words. Only those addresses mapped to the same set compete for storage in that space.
- Given an address, the hardware uses the set id bits to identify the set number, and the key bits to identify the key.
- The hardware matches the keys of the blocks already n the identified set to comparator inputs, and also provides he key of he sought address as input to all the comparators.
- Comparisons are made in parallel; in the case of a match, he block offset bits are used to index into the identified block to s l ct the particular address being referenced.

Least-recently-used

- LRU is the replacement policy most typically used. When a reference is made and is not found in a set, some block in the set is ejected to make room for the one containing the new reference.

- LRU is one of several replacement policies known as *stack* policies. These are characterized by the behavior that for any reference in any reference string, if that reference misses in a cache of size n, then it also misses in every cache of size m <*n*, and if it hits in a cache of size m, then it hits in every cache of size n> m.
- The stack distance *of* a block in this list is its distance from the front; the most recently referenced block has stack distance 1, the next most recently referenced block has stack distance 2, the LRU block has stack distance 64.
- Presented with a reference, the simulation searches the list of cache blocks for a match.
- If no match is found, then by the stack property no match will be found in any cache of a size smaller than 64, on this reference, for this reference string. If a match is found and the block has stack distance k, then no match will be found in any cache smaller than size k, and a match will always be found in a cache of size larger than k.
- Rather than record a hit or miss, one increment the kth element of a 64-element array that records the number of matches at each LRU level.
- To determine how many hits occured in a cache of size *n*, one sums up the counts of the first *n* elements of the array.
- Thus, with a little arithmetic at the end of the run, one can determine (for each set cache) the number of hits for every set of every size between 1 and 64.

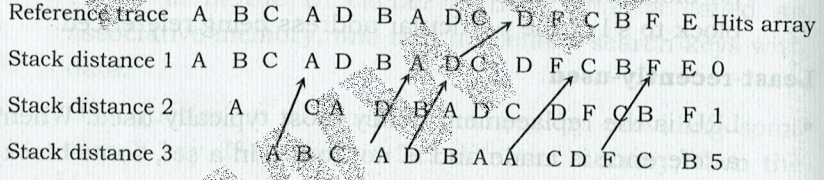


Figure 8.9: LRU Stack Evolution

- Under each reference is the state of the LRU stack *after* the reference is processed.
- The horizontal direction from left to right symbolizes the trace, re d g from left to right.
- A hit is illustrated by a circle, with an arrow showing the migration of the symbol to the top of the heap.
- The "hits" array counts the number of hits found at each stack distance.
- Thus we see that a cache of size 1 will have a hit ratio of $\frac{0}{15}$, a cache of size 1 will have a hit ratio of $\frac{1}{15}$, and a cache of size 3 will have a hit ratio of $\frac{6}{15}$.
- In one pass one can get hit ratios for varying set sizes, but it is important to note that each change in set size corresponds to a change in the overall size of the entire cache.

DISCUSSION EXERCISE

1. Write short notes on GPSS. Explain at least 5 GPSS block-diagram symbols.
2. What do you mean by Action Times and Succession of Events.
3. Design a supermarket simulation model using GPSS symbols.
4. Design a Telephone System simulation model using GPSS symbols.
5. How do you model continuous system? What is the concept of CSMP language and describe different types of statements used in CSMP with a sample CSMP program for Automobile wheel suspension problem.

6. What do you mean by GPSS? Explain with the example a manufacturing shop model.
7. What do you mean by feedback system? Explain with the help of modelling autopilot with the related CSMP program.
8. What do you mean by GPSS? Explain a manufacturing shop model with its GPSS block and code. Assume necessary data yourself.
9. Discuss model of CPU simulation.

❑❑❑

LABORATORY WORK

For Simulation and modeling laboratory works students must focus on every problems they have learned in different chapters. It is not restricted to any specific tool. In this subject different programming languages are used for some problems whereas some modeling packages are also used. Here I focused on modeling using MatLab and GPSSworld for different problems. For programming students can use C, C++, Java, Python or any high level programming language.

Some Lab related problems are:

1. Simulate Mid Square Method for random number generation in any programming language or simulation tool.
2. Simulate Linear Congruential Method for random number generation in any programming language or simulation tool.
3. Simulate KS Test in in any programming language or simulation tool.
4. Simulate Chi square Test in in any programming language or simulation tool.
5. Calculate the value of PI using Monte Carlo Method
6. Simulate random walk problem or a drunkard problem in any programming language or simulation tool.
7. Simulate differential and partial differential equations in simulation tool such as MatLab
8. Simulate discrete event simulation of an queuing system and calculate different long run measure of performance.
9. Simulate Discrete simulation language in any GPSS packages.
10. Develop different models in GPSS such as manufacturing shop model.

1. MatLab Basics:

Install MATLAB try these basic commands before using simulation tools.

If you type in a valid expression and press Enter, MATLAB will immediately execute it and return the result, just like a calculator.