Simulation Modelling with Pascal

Ruth M. Davies
Department of Accounting and Management Science
University of Southampton, UK

and

Robert M. O'Keefe
Department of Decision Sciences and Engineering Systems
Rensselaer Polytechnic Institute, Troy, New York, USA

PRENTICE HALL
New York • London • Toronto • Sydney • Tokyo

First published 1989 by
Prentice Hall International (UK) Ltd,
66 Wood Lane End, Hemel Hempstead,
Hertfordshire, HP2 4RG
A division of
Simon & Schuster International Group

Printed and bound in Great Britain by
A. Wheaton & Co. Ltd, Exeter

Library of Congress Cataloging-in-Publication Data

Davies, Ruth M., 1948–
Simulation modelling with Pascal
Ruth M. Davies, Robert M. O'Keefe.
p. cm.
Bibliography: p.
Includes index.
ISBN 0-13-811571-0
1. Digital computer simulation.
2. Pascal (Computer program language)
I. O'Keefe, Robert M., 1958– . II. Title.
QA76.9.C65D39 1988
001.4'34--dc19 87-34159
CIP

British Library Cataloguing in Publication Data

Davies, Ruth M.
Simulation modelling with Pascal.
1. Digital computer simulation 2. PASCAL
(Computer program language)
I. Title II. O'Keefe, Robert M.
001.4'24 QA76.9.C65

ISBN 0-13-811571-0
ISBN 0-13-811563-X Pbk

1 2 3 4 5 92 91 90 89 88

ISBN 0-13-811571-0
ISBN 0-13-811563-X PBK

Contents

Preface

This book gives a comprehensive introduction to discrete-event simulation. It describes and compares coded case studies for the three most popular methods of structuring simulations: the three-phase approach, event scheduling, and the process view. It provides, moreover, the statistical basis for sampling theory, experimentation, and variance reduction. Techniques for visual output are developed and more complex simulation concepts are discussed including: the role of simulation in decision-making, and the use of advanced computing.

There is a simulation package in Pascal which has the facilities of a commercial simulation package, but nevertheless can be tailored to suit any specific requirements. The appendices include the entire package, together with documentation and details on how to get it implemented with a number of specific systems.

Exercises at the ends of the chapters test comprehension of the most important concepts and enable the reader to gain confidence in their application. We have also provided an annoted biography which will be particularly helpful to readers who want to study aspects of the subject in more depth.

The idea for this book arose when we were both PhD students at the University of Southampton and wanted a text that explained in *parsimonious* detail how to develop and experiment with a discrete-event simulation model represented as a computer program. Having failed to find a suitable text, we have now written an (essentially practical) book which aims to provide this information.

PROSPECTIVE READERS

In the USA, this book may serve as the basis for a complete one-semester or one-term course in simulation. In the UK, it may be used as the basis for a whole-year course in simulation, or provide the text for half or part of a course. Such courses are typically part of the curriculum in management science, industrial engineering, and operational research departments.

They are sometimes also provided in statistics, mathematics, computer science, and various different engineering departments. This book is particularly useful for students who are familiar with Pascal.

Outside the academic community, we envisage this book as being useful to others as follows.

(*a*) For those with no prior knowledge of discrete-event techniques but who need to develop a simulation program, this book teaches the fundamental simulation concepts and provides a package for use.

(*b*) Those experienced in developing simulations with one or more different simulation packages, but who want to model a system which needs some particular modelling technique or visual output (which their packages cannot easily provide), can use this book to understand how to tackle the problem and they will be able to adapt and apply the Pascal package to solve it.

(*c*) Those who are interested in the application of simulation techniques (such as clients in simulation projects who want to understand the project and ask the right questions) can read selected chapters of this book.

Case studies

We use two cases studies extensively throughout the book: one is concerned with the use of resources by patients in a hospital system, and the other with machine breakdown in a manufacturing system. The book gives working simulation programs, both with and without visual output, for the case studies. Two further case studies, which describe a bank queueing system and a traffic light system, provide material for exercises in many of the chapters.

Structure

The first five chapters of the book provide a basic core of knowledge covering: simulation methodology, the Pascal package, sampling, and the collection and analysis of results. Chapter 6 uses this material to provide three-phase and event-scheduling simulations for the two case studies.

Chapters 7–14 contain more advanced material. The modelling of more complex systems is described in Chapters 7 and 11 while Chapter 8 is concerned with experimentation and variance reduction. Chapters 9 and 10 show how to provide simulations with visual output and the use of the

process view approach is explained in Chapter 12 with examples. Chapter 13 discusses simulation as part of the wider decision-making process and Chapter 14 describes more advanced simulation techniques and packages. Some of these later chapters can be read independently although they are all related: Chapter 10, in particular, depends upon Chapter 9, and Chapter 11 builds upon material given in Chapter 7. Examples of how to read this book for readers with specific requirements are as follows.

(*a*) Basic material: Chapters 1–6, and perhaps Chapters 7 and 8.
(*b*) Basic material with visual output: (*a*) plus Chapters 9 and 10.
(*c*) Basic material with more advanced concepts and programming: (*a*) plus Chapters 7 and 11.
(d) Basic material with the wider uses of simulation: (*a*) plus Chapters 13 and 14.

Those wishing to learn about discrete-event simulation without using or understanding the Pascal package should read Chapters 1, 2, and 4 (excluding the paragraphs at the end of some sections which explain how the concepts are implemented in Pascal), Chapter 5 (excluding section 5.3), Chapters 8 and 9 (excluding sections 9.3 and 9.4), Chapter 12 (excluding section 12.4), and Chapters 13 and 14.

Software

Proper software development, particularly stepwise refinement and modularity, is emphasized throughout. We have been careful to provide software tools and methods which can be used for advanced work as well as for the more simple textbook examples.

We chose Pascal because, not only is it used in the majority of undergraduate programming courses around the world, but also it is well-structured, strongly typed, and provides a linked record structure which is particularly useful for manipulating lists in simulation programs. Even so, most of the ideas and even much of the code contained here could equally well be expressed in FORTRAN, C, Ada, or any other procedural algorithmic language.

The Pascal package, entitled Pascal_SIM, mostly follows the ISO standard for Pascal with a few minor exceptions (see Appendix A). We have developed the package from some Pascal routines written by John Crookes at Lancaster University; we considerably changed and rewrote these while at the University of Southampton. We have further refined and simplified them for teaching to students and for presentation in the book.

Appendix A provides comprehensive documentation of the package whose complete coding is given in Appendix B. Appendix C shows how to implement the package in various different versions of Pascal.

Prerequisites

The only prerequisites for a course based on this book are: a previous course in Pascal preferably including dynamic memory allocation and pointers, and a grounding in basic probability theory and statistics.

Supplementary material

There is both a teacher's manual providing worked exercises and advice on the use of the simulation package and an IBM disk providing listings of: the Pascal_SIM package, the case study simulations and answers to exercises. The manual and the disk are available to adopters of the book. Other readers who are interested in obtaining a disk of the Pascal_SIM package, should contact one of the authors directly by post or through the publishers.

ACKNOWLEDGMENTS

A number of people have provided help with this book, to whom we are both grateful.

(*a*) John Crookes of Lancaster University, who generously gave us his programs and started us on the track of using Pascal for simulation modelling.
(*b*) Peter Bell and Paul Kirkpatrick of the University of Western Ontario for allowing us to use the example OPTIK screen shown in Chapter 9.
(*c*) Dick Nance and Osman Balci of Virginia Polytechnic Institute and State University, who provided many pearls of wisdom, some of which have slipped into Chapters 12 and 14.
(*d*) Huw Davies of the South Bank Polytechnic, who did the variance-reduction work on the repair shop case study explained in Chapter 8.
(*e*) Our former students from the South Bank Polytechnic, Kent University, and Virginia Polytechnic Institute who have provided case studies and have used and criticized the material in the book.
(*f*) Tim, Peter, and Susan, who were supportive throughout, and Jackie, who was always there.

R.M.D.
R.M.O.K.

Notation

Throughout the book the Pascal code has been presented in the following format.

(a) All complete programs and extracts from programs shown in figures and appendices are in standard `computer print-out typeface`.

(b) Within the text all Pascal code, apart from predefined Pascal_SIM words, is in *italics*. Predefined Pascal_SIM words are in **bold** typeface to distinguish them from the commentary and the rest of the Pascal coding.

1 Simulation Modelling

The word *simulation* is a generic term describing many different types of activities including: role playing in a social psychology experiment, complex video games, and scale models built by engineers to describe the behavior of bridges or aircraft. When the word is used by computer scientists, statisticians, and management scientists, they normally refer to the construction of an abstract model representing some system in the real world. The simulation describes the pertinent aspects of the system as a series of equations and relationships, normally embedded in a computer program.

Typically, we wish to develop and experiment with a model in order to address some problem. We use a model, rather than experiment with the real world system, for one of three main reasons.

(*a*) The system as yet does not exist. Simulation might, for example, be used for planning a new production facility or a new hospital.
(*b*) Experimentation with the system is expensive. Modelling may, for example, indicate whether it is wise to spend a large sum of money on new equipment.
(*c*) Experimentation with the system is inappropriate. A good example of this is disaster planning where the hospital, police and ambulance service need to plan for major accidents.

1.1 MODELLING

The identification of the system problem will, in turn, lead to the definition of *objectives*. These may be very specific, such as:

(*a*) to determine whether an additional server in a shop will keep the queue length down to a certain level,

or on the other hand be much more vague, such as:

(*b*) to understand the population dynamics of pond life.

In modelling any system, a process of selection has to be made whereby some elements of the system are modelled and some are assumed to be unimportant or irrelevant in the context of the objectives. A model thus not only embodies objectives but also *assumptions*. These should be stated clearly as they may have to be questioned at a later stage in the modelling process.

If we regard the simulation model as a black box, it is clear that the simulation must provide *output* or *responses* which are pertinent to the questions asked and the objectives that have been set. This black box must be fed with *input* to provide information about the state of the system being modelled. Some input variables are controllable by those that manage the 'real life' system. These are called *decision variables*. They may be changed to see the effect on the *output* or *response variables*.

An *objective function* expresses the objectives as a mathematical function of the output measures. If the objectives of a simulation study can be expressed as an objective function, it may be possible to determine an optimum strategy by a series of carefully organized simulation experiments. If, on the other hand, the objectives are vague and the definition of an objective function is not feasible, such as is frequently the case with social systems, the simulation may be used in an exploratory way. It might, for example, be used to examine the consequences of different decisions or policies.

Simulation is thus a descriptive tool, allowing us to experiment with a model instead of the real system. The purpose of building the simulation and hence its use, generally falls into one of three categories as follows.

(*a*) *Comparison*. A comparison of simulation runs can be used to assess the effect of changing a decision variable. The results of the different runs can then be evaluated in terms of the objectives.

(*b*) *Prediction*. A simulation may be used for predictive purposes to determine the state of the system at some future point in time, subject to assumptions about how it behaves now and how it will continue to behave.

(*c*) *Investigation*. Some simulations are developed to provide an insight into the behavior of the system, rather than to perform detailed experimentation. It is of interest to see how the simulation behaves and reacts to normal and abnormal stimuli.

This classification of the simulation objectives helps to determine the type of analysis and experimentation that will be needed. However, the distinction between these forms of investigation is not always clear cut. Sometimes, for example, an exploratory simulation will be followed by a more detailed statistical analysis, comparing or predicting the effects of changing different variables.

1.2 DIFFERENT TYPES OF SIMULATION

There are three types of model-building simulation: *statistical, continuous*, and *discrete*. These, however, are related not only because elements of statistical simulation appear in the other two, but also because there are combined simulations employing both continuous and discrete techniques. This book will be concerned solely with *discrete* simulation but a brief discussion of the other two types will help to clarify its characteristics.

1.2.1 Statistical simulation

A system that is subject to random processes is said to be *stochastic* whereas one that is not is *deterministic*. Furthermore, a system that does not vary with time is *static* whereas one that varies is *dynamic*. A *statistical simulation* describes systems which are both stochastic and static and is used to estimate values that cannot be easily deduced mathematically. This type of simulation is sometimes also called a *Monte Carlo simulation*.

In order to introduce the necessary concepts we shall consider the relatively simple problem of estimating the area of an amorphous shape. Suppose, for example, we want to estimate the size of the shaded area in Fig. 1.1, which may represent a lake, field or the outline of a city. Direct calculation of the size is not possible because of the irregularity of the shape, although a numerical approximation can be obtained by breaking the shape down into smaller regular shapes such as rectangles and circles.

In order to solve this problem using a statistical simulation, a rectangle is drawn on the map (see Fig. 1.1) so as to enclose the shaded area. Suppose it has a length of 100 units and a height of 50; hence the area is 5000 units. The rectangle is mapped onto the X and Y co-ordinates such that the bottom left-hand corner has the co-ordinate (0, 0) and x is in the range 0–100 and y in the range 0–50.

An arbitrary pair of co-ordinates will be either inside the irregular area or outside. (Being exactly on the boundary is equivalent to being

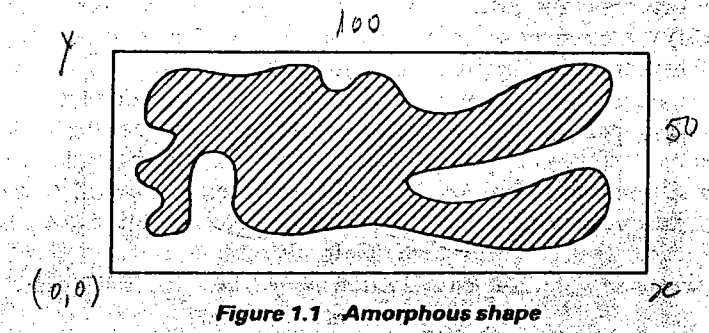

Figure 1.1 Amorphous shape

inside the area.) Thus if a pair of co-ordinates is chosen at random, it will result in a 'success' (inside the area) or 'failure' (outside). If a number of random co-ordinates are generated, the ratio of successes to the sample size will reflect the ratio of the shaded area to the area of the rectangle. Thus, if A is the true size of the shaded area and m is the number of successes in n generations of random pairs then:

$$\frac{m}{n} = \frac{A}{5000} \quad \text{as } n \text{ tends to infinity}$$

A random variable which is given the value 1 if there is a success, and 0 if there is a failure, has a Bernouilli distribution. Its parameter, p, is estimated by m/n as n tends to infinity.

Each random number in the pair of co-ordinates is generated independently and in the appropriate range. Readers interested in experimenting with this example can try sampling numbers from random number tables (which can be found in books of statistical tables).

In general, statistical simulations sample random numbers and derive results based on the values of the numbers sampled. The results themselves are thus also samples from distributions.

In more complex examples, random numbers are sampled from probability distributions rather than from a uniform range of numbers. This process is called *distribution sampling*. Chapter 4 explains the computer techniques for random number and distribution sampling.

Statistical simulation techniques are used widely in *risk analysis* for assessing the risks and benefits of different, and often very expensive, decisions.

1.2.2 Continuous simulation

Continuous simulation is used to model systems which vary continually with time. The systems are dynamic but may be either deterministic or stochastic.

Consider modelling the operation of an electric kettle. When switched on, the element heats the water until it reaches a certain temperature. This causes the kettle to turn itself off. Figure 1.2 shows a plot of the water temperature against time. The increase in water temperature, and its slower decrease following the kettle turning itself off, is a continuous process which can be represented in a continuous simulation as a number of differential equations. This approach would be useful if we were modelling the kettle with a view to comparing the operations of various types of kettle.

Continuous simulation is used extensively where feedback occurs in a

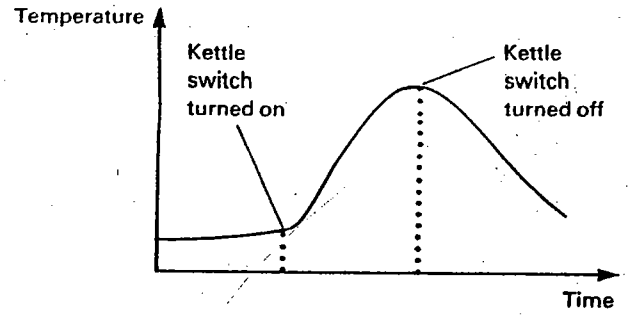

Figure 1.2 Action of kettle

system. A good example of this is the use of a thermostat with two settings to control room temperature. With the heating on, the temperature in the room increases until the upper setting on the thermostat is reached, at which point the heating switches off. The temperature will then decrease until the lower setting on the thermostat is reached, at which point the heating will be switched back on. Such continuous simulation models, often composed of scores of feedback loops and hundreds of differential equations, are used extensively in mechanical, production, and electrical engineering.

A variation on continuous simulation called *systems dynamics* was developed by Forrester and his co-workers at MIT in the early 1960s, for use in problems that can roughly be called socio-economic. Systems dynamics has been used to model urban growth, industrial production, fishing regulations, ecosystems, and many other systems. For an introductory text, see Roberts *et al.* (1983).

1.2.3 Discrete simulation

If, in the kettle example, (see Fig. 1.2) the intermediate water temperature is of little interest, the action of the kettle can be considered as two discrete events: switching the kettle on, and the kettle switching itself off. If, for example, we were modelling the activities of a kitchen we might describe the kettle in this way.

Discrete-event simulation is concerned with the modelling of systems that can be represented by a series of events. The simulation describes each discrete event, moving from one to the next as time progresses. The systems modelled are dynamic and, almost invariably, stochastic.

Consider, for example, a simple inventory system for a single product where, each week, the operator must make a single decision about how

much to order. We can identify three events in this system: the placing of an order, the arrival of an order, and a sale.

If the demand for the items and the time for an order to arrive from the supplier are assumed to be deterministic, then the size of each order can be calculated analytically. If, however, the variables are regarded as stochastic (i.e., they are sampled from probability distributions) the problem is more complicated and discrete-event simulation may help determine the best order size.

In describing stochastic systems, distribution sampling determines the operation of the system and the length of the activity times. Thus, the output measures of a simulation are normally samples from statistical distributions. Clearly several samples will be needed to provide good estimates of distribution parameters.

1.3 TWO CASE STUDIES

In setting up a simulation study in an organization, a description of a system and of the problems leading to the need for a simulation study may have to be built up through interviews with staff, the research of papers showing, for example, minutes of committee meetings, and data collection exercises. The description should provide the information to decide on: the objectives of the study, some initial assumptions, decision variables, and the key events to be described in the simulation. The assumptions may be relaxed at a later stage in the modelling process if the model is not considered to be sufficiently detailed. As a rule of thumb, models should be as simple as possible to produce results within the requirements for accuracy.

The research together with the filtering and simplification process can, in practice, be extremely time consuming. In providing the following case study descriptions, we are assuming that this background work has already been accomplished.

1.3.1 Hospital

Patients are central to the use of expensive resources in health care. The activities of patients in a hospital environment are complex, however, and many different facilities are used during the course of treatment. Simulations can be used to model the activities of patients to enable clinicians to plan resource use.

The hospital case study is based on extremely common activities

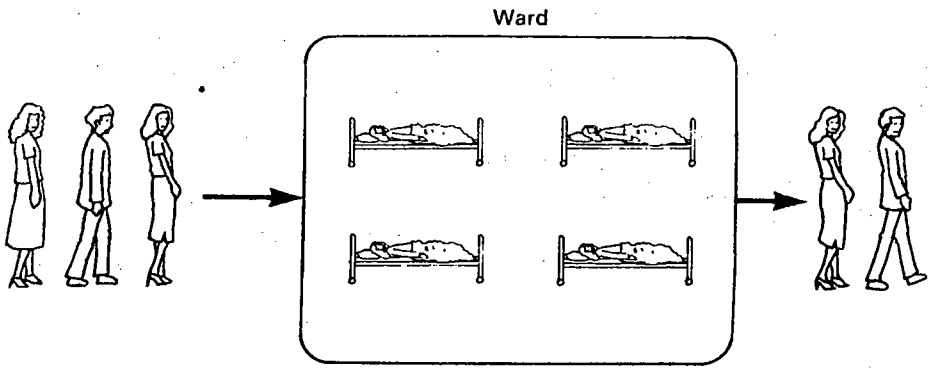

Figure 1.3 Simple hospital system showing queue for ward with four beds

which are probably familiar to all readers. The number of beds in a hospital ward is a constraint on the use of hospital facilities. Patients are admitted to a ward and, following treatment, are discharged. The purpose of designing this simulation is to examine the effect of different resource levels (e.g., number of beds) on the service given to patients (measured by the waiting time for admission, or by the length of the waiting list).

Figure 1.3 shows that patients, identified by a doctor as needing treatment, are put on a waiting list. (In the simulation, new arrivals may be sampled from a statistical distribution and put in a queue.) When a bed is available and there is a patient in the waiting list, the patient is admitted to the ward and a bed is committed until discharge. The bed is then released and becomes available for another patient.

This model is very simple and it is unlikely to be sufficiently detailed for any practical purposes. Therefore, we are going to look at a slightly more complex system in which some patients require operations and some do not. They join different waiting lists, have different average arrival rates and lengths of stay. Priorities now have to be assigned between the two groups waiting for the available resources. In this example, those who do not require operations (assuming they are the emergencies) are given priority for the available beds.

A further complexity is that the operating theatre may be effectively constraining the number of admissions to wards. Those patients requiring operations are put in a waiting list for the operating theatre *after* they have acquired a bed. There is one operating theatre which is sometimes shut and sometimes open. Figure 1.4 shows patients being operated on, and returned to the ward after an operation. Experimenting with the model can show whether increasing the times at which the theatre is available, or

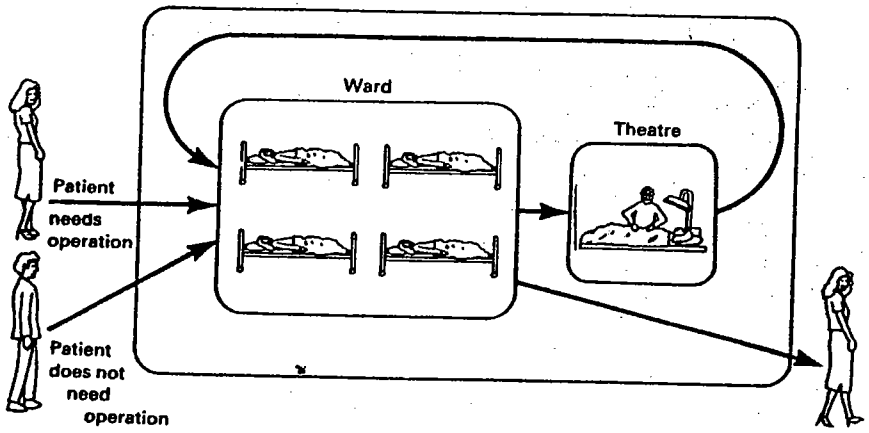

Figure 1.4 Hospital system showing queue for ward with four beds and one operating theatre

changing the number of beds, can increase the throughput of patients in the ward.

1.3.2 Repair shop

Perhaps industrial engineers are the most frequent users of simulation techniques. The technique has been used to model entire production processes and individual aspects of production such as job shop scheduling, maintenance provision and the operation of repair shops.

The example presented here is a classic application area for simulation: modelling machine breakdown. Within many production environments, machine breakdowns occur all too frequently. Often staff will be employed simply as repair mechanics, with the sole task of repairing broken machines as quickly as possible. The objective of the management is to minimize *downtime* by making sure that broken machinery is back in operation as soon as possible. However, against this objective must be offset the cost of providing both repair support and the equipment necessary for repair work to proceed.

Figure 1.5 shows the activities of a typical machine in the repair shop. When a machine breaks down, the mechanic removes all covers and also

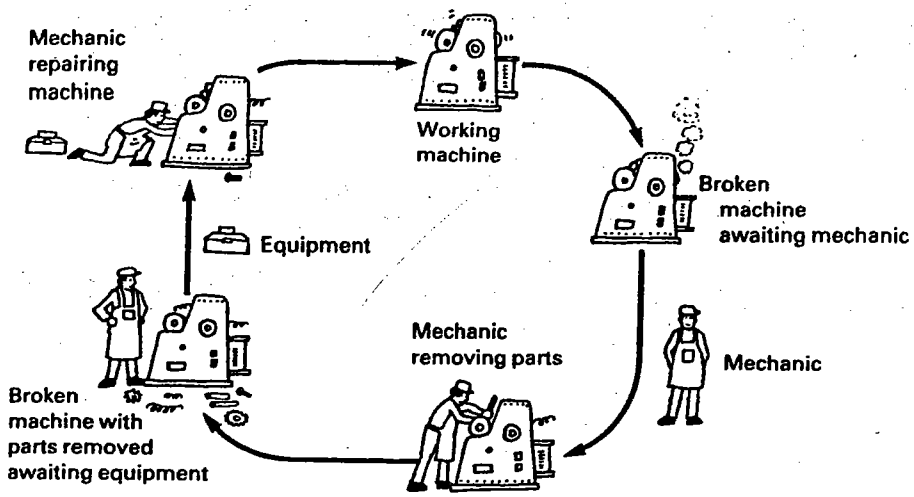

Figure 1.5 Life cycle of machine in repair shop

any material stuck in the broken machine, prior to obtaining the equipment necessary for the repair of the machine.

The problem is to determine how the numbers of mechanics and items of equipment provided will affect machine downtime and hence the productive capacity of the repair shop.

Unlike the hospital system, which the patients entered and left, the production system here is self-contained – nothing flows into or out of the system. In reality, raw or partially finished materials may be processed by the machines, but the flow of these is considered to be beyond the bounds of the system. In effect, it is assumed that there is always enough material for the machines to be busy if they are not broken.

Again, as with the hospital system, generalizations are being made: we assume that all the machines are identical, we make no distinction between different types of equipment which may be as diverse as cranes and spanners and, furthermore, we assume that the time the mechanics take to transport the equipment to a broken machine is negligible.

1.3.3 Comparison of two systems

In the hospital the main objects are patients who are processed or served by the other objects in the system whereas in the repair shop the machines are the main objects. Although they go through a number of activities,

they remain static in the system. The former approach is called *material* or *customer oriented*, whereas the latter is *machine oriented*. However, the distinction is largely academic and is not of crucial importance in the simulation design.

Another aspect of the system is the time scale to which the system operates. We will assume that the repair shop works eight-hour shifts and, if any machines are in the process of being repaired at the end of the shift, they are returned to working order ready for the next shift. We are thus modelling a system that has distinct starting and finishing times and always starts off in a cold state, whereas hospital wards usually operate continuously, day and night. Furthermore, the time for a repair takes a matter of minutes, whereas the time between the admission and release from hospital usually takes a matter of days or even weeks.

The differences between the two examples, as indicated above, are worth emphasizing as follows.

Hospital case

(*a*) The system is open – transient objects flow through the system.
(*b*) The system is customer oriented – patients are the main objects under consideration.
(*c*) The system is an ongoing process – admission and release from hospital will continue into the foreseeable future.
(*d*) The activities can last for days or even weeks.

Repair shop case

(*a*) The system is closed – the number of machines remains static.
(*b*) The system is machine oriented.
(*c*) The system terminates after every eight hours.
(*d*) The activities last for minutes or hours.

Note that other systems have different combinations of these aspects. For instance, consider the service of clients in a bank or post office. The system is open and customer oriented (like the hospital), but it terminates, and activities last for a matter of minutes (like the repair shop). In later chapters, we will see how these differences affect the way in which each example is modelled.

From the descriptions of the case studies we can identify the objectives, the main assumptions that we are going to make in modelling the system, the responses from the simulation, the decision variables, and the events. These are summarized in Tables 1.1 and 1.2.

In both of these case studies our main interest is in the comparison of simulation runs rather than in either a general investigation of how the system works or else in making predictions.

Table 1.1 Summary of hospital system

Objectives
To investigate the effect of bed and operating theatre provision on patient waiting times.

Initial assumptions
The system runs continuously with no breaks for weekends or holidays.
Patients who do not require an operation are assumed to be emergency patients and are admitted in preference to those who do.

Decision variables
Number of beds.
Opening times of the operating theatre.

Responses
Patient waiting lists and waiting times.

Events
Patient arrival, admission, the start of an operation, the end of an operation and discharge.
Operating theatre opens, operating theatre closes.

Table 1.2 Summary of repair shop system

Objectives
To investigate the effect of mechanic and equipment provision on machine downtime.

Initial assumptions
A shift starts with no machines working and lasts until all machines are returned to working order after the end of an eight-hour shift.
There is always enough material to keep the working machines busy.
All the machines are identical.
The time taken for a mechanic to travel to a machine or to transport equipment to a machine is negligible.

Decision variables
Number of mechanics.
Number of sets of equipment.

Responses
Machine downtime.
Utilization of mechanics.
Utilization of equipment.
Post-shift working.

Events
Machine breakdown, start of equipment removal, end of equipment removal, start of repair, end of repair.

1.4 SUMMARY

Simulation involves building a model of a system to meet one or more *objectives*. The level of detail and the *assumptions* that are made clarify these objectives. We have classified the models as: *comparative*, *predictive* or *investigative*.

There are three types of model building simulation: *statistical*, *discrete*, and *continuous*. Statistical simulation provides estimates of distributions. Continuous simulation is used to model systems which vary continuously with time, whereas discrete simulation is used to model systems which are assumed to change at discrete points in time.

A discrete system can be either open or closed, customer or machine oriented, continuing or terminating. A simulation may model a few seconds, many months of activity or many years.

This book only deals with discrete simulation, although the sampling facilities presented can be used within a statistical or continuous simulation.

1.5 EXERCISES

1. Get some centimetre graph paper and draw a square 10 cm by 10 cm. Draw a shape within the square and estimate its size by counting the millimetre squares. Using random number tables from the back of a statistics book, and selecting these in pairs, estimate the area of the shape by the method of statistical sampling. How many samples do you think you will have to take before you get within 10% of the more accurate estimate?

2. The first, second, third, fourth, and fifth prizes at the hospital raffle are: a radio worth £200, a food blender worth £60, a bottle of whisky worth £30, a bottle of wine worth £3, and a box of chocolates worth £1.50, respectively. Altogether, 10000 tickets are sold, of which you buy 10. Use statistical sampling methods (repeating the experiment 50 times) to estimate your expected winnings. Compare this to the theoretical value and comment on the results.

3. Identify the important discrete events in the following systems.
   (*a*) The investigation of queues in a petrol station.
   (*b*) A change in the train timetables on a railway network.
   (*c*) The provision of terminals on a main-frame computer system.
   (*d*) A decision about the number of waiters required in a restaurant.
   (*e*) A production line for making sliced bread from the basic ingredients of flour, yeast, water, salt, and additives.

4. Read the case studies in Appendix 1.A of this chapter and answer the following questions for each of the *bank system* and *traffic light system*.
   (*a*) State the objectives of the study. Identify the decision variables and the responses of interest to those commissioning the study.
   (*b*) If this system is to be described as a discrete-event simulation, identify the discrete events that are likely to be of interest. Classify the model as predictive, comparative, or investigative.
   (*c*) List any assumptions that you will need to make in order to model the system.

* * *

1.A MORE CASE STUDIES

These case studies will be used in the exercises throughout the book. Therefore, readers will find that they will need to refer back to them frequently.

1.A.1 Bank case study

A local branch of the Royal Interest Bank in Savingston has four tills, one of which (Till 4) is only open when the bank is very busy. Till 2 is reserved for customers who wish to withdraw money by cheque but these customers may, if they wish, go to any of the tills. The transactions that are permitted at any till, except Till 2, include money deposits, foreign currency transactions, and the payment of bills. The time taken to serve different customers at the tills varies considerably. Those depositing cash from shops and small businesses, in particular, take a long time.

The bank manager is disturbed by complaints from customers about long waiting times, especially when they queue behind someone depositing large amounts of cash. They also find that because the waiting space is cramped when the bank is busy, it is difficult to identify the shortest queue. Furthermore, if they want to join a queue at the end of the bank, they have to push through the nearer queues to get there.

The bank manager thinks that a one-queue system, in which customers wait in one queue and are served at the first available till, might solve these problems and is prepared to make some alterations to the bank to implement it. However, he is concerned that a one-queue system would be less personal and would disadvantage those who simply want a cheque transaction and can now join a fast moving queue. He also points out that the additional time it will take for customers to move from the front of the queue to the service till, which he estimates would take 10 s on average, might offset the benefits. He would like help in resolving this question.

Data on arrival and service times were collected at three different

Table 1.3 Customer arrival data from bank survey

Time period in secs	*Observed frequency*
0 – 10	60
10 – 20	42
20 – 30	23
30 – 40	18
40 – 50	14
50 – 60	10
60 – 70	7
70 – 80	5
80 – 90	8
90 +	9
	196

Table 1.4 Service time data from all tills in bank

Time period in secs	*Observed frequency*
0 – 25	32
25 – 50	43
50 – 75	40
75 – 100	31
100 – 125	19
125 – 150	8
150 – 175	8
175 – 200	3
200 – 225	8
250 +	7
	199

times of day over a total period of 110 min. These were combined to give the interarrival distribution shown in Table 1.3 and the service time distributions in Tables 1.4 and 1.5. The survey showed that one third of the arriving customers joined the queue for Till 2.

A further sample survey showed that the arrival rate varied with the time of day (see Table 1.6). There was also some variation with the day of the week but this was not thought to be significant.

1.A.2 Traffic light system

Figure 1.6 shows the pattern of traffic lanes on Oneway Road at a busy junction. The traffic lights vary regularly from red to green (21 s) and back

Table 1.5 Service time data from Till 2 in bank

Time period in secs	*Observed frequency*
0 – 25	16
25 – 50	14
50 – 75	6
75 – 100	13
100 – 125	7
125 – 150	2
150 – 175	0
175 – 200	3
200 – 225	2
250 +	2
	65

Table 1.6 Average arrival rate of bank customers with time of day

Time in hours	*Average arrival rate*	*No. tills open*
9·30 – 10·30	51	3
10·30 – 11·30	73	3
11·30 – 12·30	109	3
12·30 – 13·30	213	4
13·30 – 14·30	84	3
14·30 – 15·30	114	3

to red (29 s) again. The traffic is particularly heavy in the rush hour with a build up of long queues at the junction. A bypass, to be finished in three years time, is expected to take some of the pressure off the road and the council are interested to know to what extent the traffic flow on Oneway Road at this junction will be improved.

Data was collected during 67 cycles of the traffic lights during the evening rush hour. Most of the vehicles counted were cars but there were a few bicycles, lorries, buses, and motor bikes. The total number of arrivals in each phase are shown in Table 1.7.

The ratio of traffic in each lane remained fairly constant at:

(*a*) traffic leaving major road 73·6 %
(*b*) traffic turning right 24·4 %
(*c*) traffic turning left 2·0 %

Finally, those collecting the data observed that when the number of vehicles that were waiting, in any lane, exceeded 15 when the traffic lights changed from red and amber to green, then on average, 15 vehicles passed through the lights in that lane before the lights became red again.

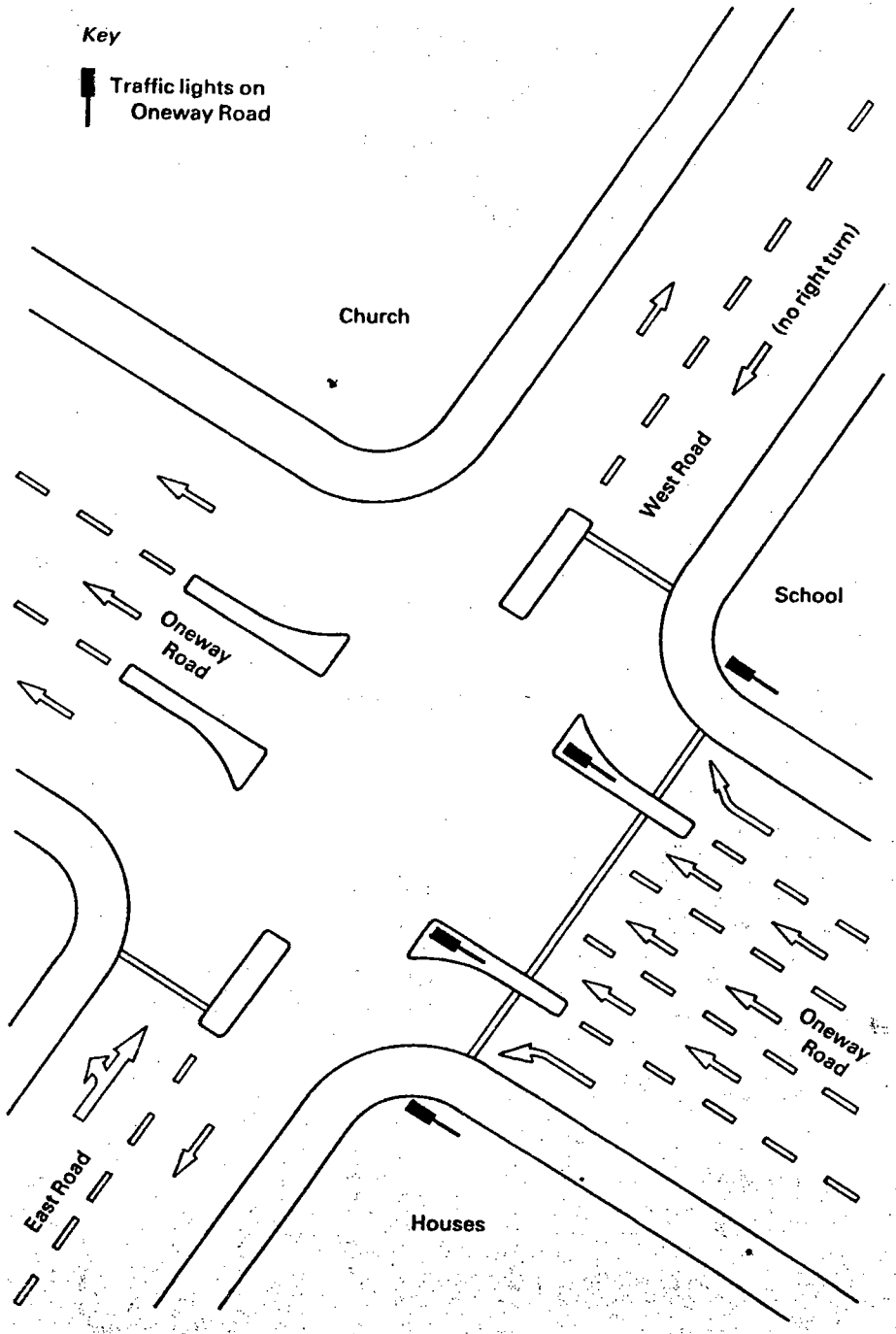

Table 1.7 Total number of vehicles arriving in each of 67 cycles of traffic lights, in the centre lane, during evening rush hour

No. vehicles approaching lights	*Frequency*
14	1
15	1
16	1
17	1
18	2
19	3
20	5
21	1
22	2
23	7
24	5
25	2
26	3
27	2
28	5
29	5
30	6
31	3
32	2
33	4
34	1
35	4
36	1

Figure 1.6 (opposite) Layout of traffic lanes on Oneway Road at a busy junction; traffic lights on East and West Road not shown but co-ordinate with those on Oneway Road

2

Simulation Methodology

Simulation packages incorporate mechanisms for performing the discrete events in the right order. Tocher (1962) was one of the first people to structure simulations in this way and although other approaches have evolved, they all have many common elements. It is important, therefore, to understand the concepts and the corresponding terminology before attempting to develop a model.

In this chapter, we shall define and explain the terminology used throughout this book. We shall also show how systems can be described by diagrams and modelled using two different approaches. All new concepts will be illustrated by using the case studies outlined in Chapter 1. They are referred to as: the *simple hospital system* (Fig. 1.3), the *hospital system* (Fig. 1.4), and the *repair shop system* (Fig. 1.5).

2.1 TIME

In Chapter 1 a *discrete-event simulation* model was defined to be one in which changes in the state of the model, called *events*, are assumed to take place at discrete points in time. Each point in time at which one or more events take place is called a *time beat*.

The measurement of *time* in a simulation run corresponds to appropriate units of time in the actual system, whether minutes or millenia. The time period over which the simulation is to run is referred to as the *duration* of the simulation. However, the time the computer takes to run the simulation is largely dependent on the number of events that have to be performed and bears little relation to the duration of the simulation.

A simulation program, starting at time zero, performs all events in the order in which they occur, advancing from one to the next until:

(*a*) there are no more events to perform,
(*b*) the time of the next event to be performed exceeds the maximum time set for the duration of the simulation, or
(*c*) some terminating event is encountered.

Example

(1) In the simple hospital system, time could be measured in days. If it were measured in parts of a day or hours, however, the simulation would give a more accurate picture of bed availability and the use of the ward. Hours, or even smaller subdivisions of time, are necessary for looking at the use of operating theatres in the hospital system or for simulating the repair shop system.

The duration chosen for the hospital system is arbitrary because we should normally expect the activities in this system to continue without a break. However, the repair shop system only operates during working hours and therefore the natural duration to choose is the length of the working day.

2.2 ENTITIES AND RESOURCES

The objects, or individuals whose activities are being modelled, are represented in a simulation program by *entities*, which can be individually identified by their *attributes*. The attributes invariably include a number indicating the time of the next event, sometimes called an *entity clock*.

Resources are items in the simulation which do not have characteristics but act as constraints on the activities of the entities. Resources can be thought of as being held in a receptacle or *bin* from which they are *acquired* when needed and *returned* after use.

Example

(2) In the simple hospital system, patients are entities and the *beds* are resources. The *patients*, which may be individually identified, each have an entity clock to record the time of the next event. The *beds* are assumed to have no individual characteristics. They are acquired from the bin when a *patient* starts a hospital stay and are returned at the end of that stay. In the hospital system, the *operating theatre* is also an entity.

There is an important distinction between *permanent* and *temporary* (or *transient*) *entities*. *Permanent entities* are created at the beginning of a simulation run and remain in it throughout, whereas *temporary entities* are created when they are required and disposed of when no longer needed.

Example

(3) The *patients* in both hospital systems arrive in the system from outside, and after discharge disappear to an 'outside world'. The

most natural way to describe this phenomenon is for the simulation program to create temporary entities when they are needed and to dispose of them when they leave the system.

On the other hand, in the repair shop system the *machines* are the entities. These would be described as permanent entities because they remain in the simulation throughout with no new *machines* being introduced or removed.

Entities belong to *classes* which distinguish between groups of entities.

Example

(4) In the hospital system, the *patients* who need operations, those who do not need operations, and the *threatres* are entities from three different classes.

2.3 EVENTS

An *event* occurs when something happens to an entity at a particular point in time. There are two distinct types of events:

(*a*) a *bound event* (or *B event*, sometimes called a *scheduled event*) is one whose occurrence is predictable and can thus be scheduled;
(*b*) a *conditional event* (or *C event*, sometimes called a *contingent event*) is one whose occurrence is dependent upon the fulfilment of certain conditions (e.g., the availability of certain resources).

When an entity is scheduled to take part in a bound event, the time of the bound event is written to the entity clock. There is an important type of bound event called a *feeder* whose role is to generate temporary entities. Each time a feeder generates an arrival, it creates the next entity and sets the time of its arrival.

Example

(5) The arrival of patients is an event called *patient_arrival* which is a feeder which sets the time of the patient arrivals.

The start of a hospital stay, called *start_hospital_stay*, is a conditional event because it can only take place if a *bed* is available and a *patient* is waiting for admission. However, the times of arrivals and discharges are set and scheduled by the simulation program. The event *patient_arrival* and the event of patient discharge, *end_hospital_stay*, are thus bound (or scheduled) events.

2.4 STATES AND QUEUES

Once created, an entity in a simulation may be in one of three states: *busy*, *queueing* or *idle*.

(*a*) An entity is said to be *busy* if it is scheduled to take part in a bound event.
(*b*) Entities waiting in turn for some condition to be satisfied are said to be *queueing*. The most frequently used criterion for selecting entities from a queue is *first-in-first-out* (known as FIFO). Other queue disciplines are discussed in Chapter 7.
(*c*) Entities which are not busy or queueing are said to be *idle*.

Example

(6) In the hospital system, *patients* may be waiting for admission (i.e., queueing) or in hospital (i.e., busy). When the *theatre* is not in use, it does not need to join a queue because it is unique; it is thus said to be idle.

2.5 ACTIVITIES

An entity which is busy is said to be *engaged* to an *activity*. Activities are usually started by conditional events and finished by bound events whose number and time of occurrence is scheduled in the conditional event and held on the entity clock during the course of the activity.

Example

(7) In the simple hospital system, the hospital stay is an activity which will be referred to as *hospital_stay* which is started by *start_hospital_stay* and finished by *end_hospital_stay*.

2.6 BRANCHING FROM ACTIVITIES

In a simulation, entities sometimes have a choice of routes through the system. Branching may take place after either a bound or a conditional event. There are many different criteria for choosing one branch or another. Some typical ones are as follows.

(*a*) The choice as to whether entities should go down one path rather than another is unpredictable.
(*b*) The choice of path depends on some inherent characteristics of

the entity. For example, if the entity attribute number is less than 10, the entity might take one route and if greater or equal to 10, another.

(*c*) The choice depends on resource availability or queue length at that particular point in time.

Branching is not used in the case studies introduced in Chapter 1, but will be discussed in more detail in Chapter 7.

2.7 ACTIVITY DIAGRAMS

An *activity diagram* describes the life of entities in the system and their interaction with other entities. Where temporary entities are used, the sequence of entities may be more logically represented as a flow of activities from beginning to end, called an *activity-flow diagram*, rather than as a closed cycle of activities, called an *activity-cycle diagram*, used for permanent entities.

Queues are shown by large circles and activities by rectangles. The resources are represented by smaller circles. The lines with arrows show the order in which entities engage in activities or resources are committed. The creation and destruction of entities are shown by zigzag lines.

The case studies introduced in Chapter 1 can now be more fully described as a series of activities and events, and drawn as activity diagrams.

2.7.1 Hospital systems

Figure 2.1 demonstrates how to represent the simple hospital system. The compressed arrow indicates the feeder, *patient_arrival*, which brings patients into the system. After each arrival, the time is set for the next arrival to take place. Following the arrows down the page, the first circle represents the queue of *patients* waiting for hospital admission. The rectangle represents the activity, *hospital_stay*. The events which start and finish this activity are not shown explicitly.

Figure 2.2 shows the activity flow of the hospital system. This starts with two separate streams of arrivals at the top of the page, feeding into the two queues, *q*1 and *q*2. Those patients in *q*1 are waiting to engage in the activity, *hospital_stay*, and are competing for *beds* with those in *q*2, who are waiting for hospital admission prior to having an operation. The way in which priorities are allocated to the patients in the different queues will be discussed later. Entities leaving the activity *pre_operative_stay*, must enter a queue, *q*3, until the *theatre* is available. The *theatre* is only available to this ward at particular times of the week. It is thus described as an entity in its own right, cycling through states of idleness and availability.

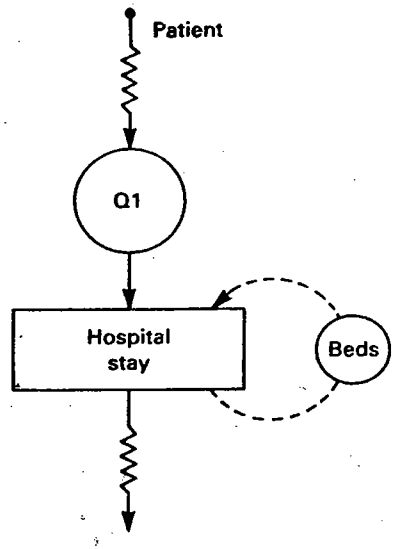

Figure 2.1 Activity diagram for simple hospital system

After their operation, patients join a queue for their *post_operative_stay*. There are no constraints on this activity and so it is called a *dummy queue*. It is usual to indicate the dummy queues in activity diagrams because:

(*a*) in many simulation languages they must be explicitly defined as queues (although this is not so in the Pascal package described in this book), and

(*b*) constraints may need to be added at a later stage when experimenting with the model.

After the *post_operative_stay* the *patient* is discharged.

2.7.2 Repair shop system

Figure 2.3 shows the activity cycle of the *machine* in the repair shop system. The most striking difference between this and the previous two diagrams is that the *machine's* life is shown as describing a complete cycle.

When the *machine* finishes the activity *work*, it must wait for a *mechanic* to be available before starting *remove*. After finishing *remove*, it then queues again until *equipment* becomes available as well as the *mechanic* before starting *repair*. Both the *mechanic* and the *equipment* are returned to the bin after *repair*.

After *repair*, the diagram shows the entity as joining another queue, *q*3, before starting *work*. In practice, the mechanics would not have to wait because the activity is unconstrained; so this is a dummy queue.

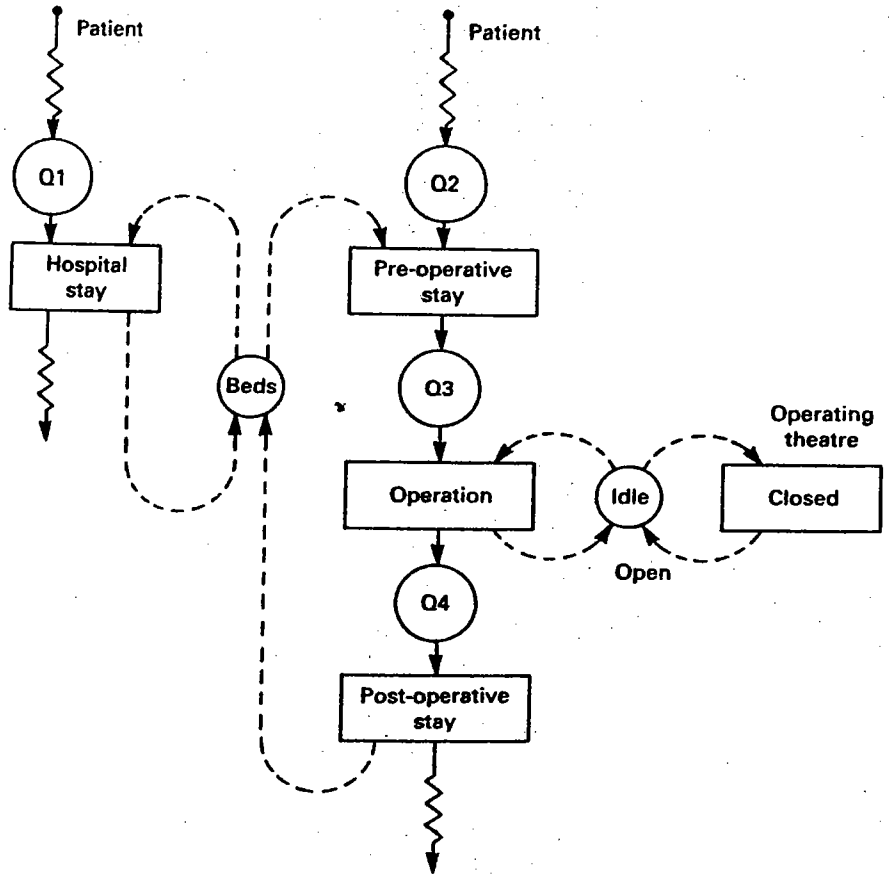

Figure 2.2 Activity diagram for more complex hospital system

The *equipment* and the *mechanics* are simply resources, in the same way as the *beds* in the previous example. The *mechanic* is reserved for more than one activity in the same way as the beds in the previous example. The activity diagram shows the involvement between the life cycles of different entities and resources in the simulation.

2.8 EXECUTIVE

2.8.1 Calendar

A calendar is either a list of entities which identify the next events to be performed or a list of events identifying the entities to take part in those

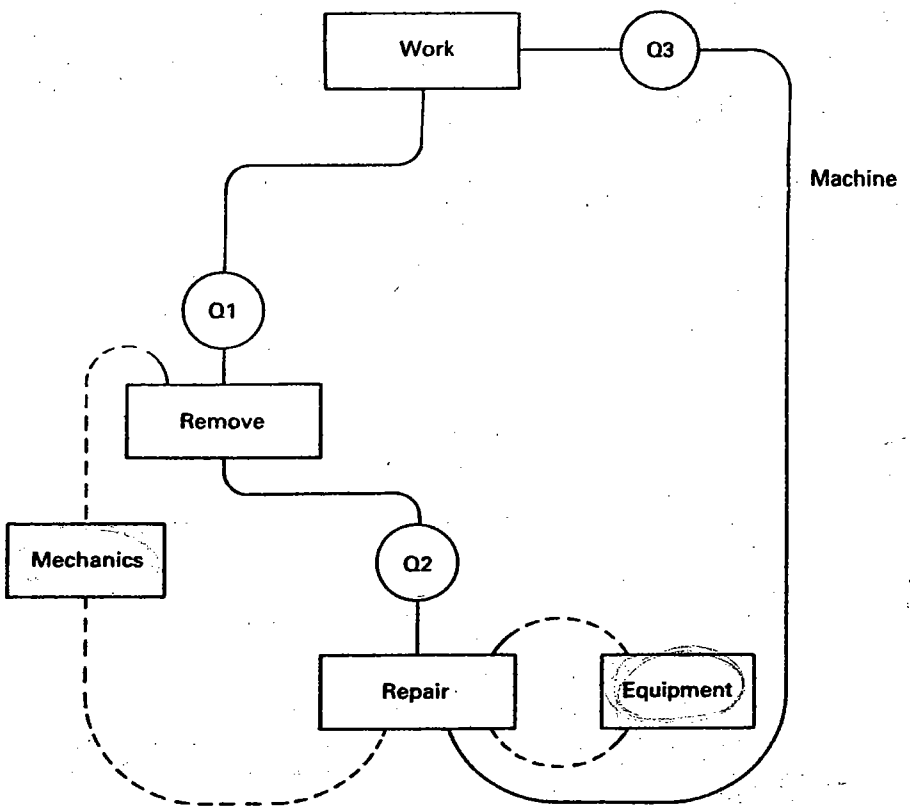

Figure 2.3 Activity diagram for repair shop system

events. This book uses the former approach. The calendar is ordered by the times on the entity clocks. Alternative names for the calendar include: the *next-event set* or the *future-event list*.

The *executive* or *time-advance mechanism* is the part of the simulation program that makes all the events take place in the correct order. The first task, or *phase*, is to advance the simulation time to the time of the next scheduled event. This is done by selecting the first entity from the calendar. The subsequent phases activate the event for that entity and then, progressively, activate each event scheduled for the same time beat. When this is complete, the simulation returns to the first phase and advances the simulation time to the time of the next event.

In order for a simulation to start working, there must be an entity in the calendar. One or more should be put in the calendar in the initialization phase of the simulation. If there is a feeder in the simulation, one of these must activate the feeder or it will not start working.

2.8.2 World views

There are three methods of structuring a simulation in common use:

(*a*) the *three-phase approach*,
(*b*) the *two-phase* or *event method*,
(*c*) the *process view*.

These different methods are called *world views*.

Although each of these three approaches can be programmed in Pascal, the *process view* has a different executive from the other two approaches. This chapter uses the three-phase approach originally devised by Tocher (1962) and the event method. Chapter 12 describes the concepts behind the process view and gives examples of its use.

In the three-phase approach, bound events and conditional events are programmed as separate procedures. The three phases of the executive are as follows:

(*a*) advance the clock to the time of the next scheduled event,
(*b*) execute all bound events due to happen at this time,
(*c*) test all conditional events and execute those whose conditions are satisfied.

Bound events and conditional events are thus written as separate and independent modules.

On the other hand, in the event method the procedures for scheduled events include all the conditional events which happen as a direct result of those scheduled events. Therefore the scheduled events have many more statements than those in the three-phase approach. The event approach consists of the following two phases:

(*a*) advance the clock to the time of the next event,
(*b*) perform the next scheduled event due to happen.

The structure of the executive is thus exactly the same as for the three-phase approach, with the omission of the third phase.

2.9 PSEUDO-CODE

The next step in designing a simulation program after drawing an activity cycle or flow diagram is to write the *pseudo-code*. The pseudo-code, like the structured English described by DeMarco (1978) used in systems analysis, is a subset of the English language written in a rigid format without any qualifiers. The language is less precise and more transparent than a Pascal program but is easy to translate to any structured programming language.


```
procedure patient_arrival; { B1 }
  begin
    < put the newly arrival patient in Q1 for admission >;
    < create the next patient >;
    < set the time of arrival of this next patient >;
  end;
procedure start_hospital_stay; { C1 }
  begin
    while < Q1 is not empty and a bed is available > do
      begin
        < admit the next patient from Q1 >;
        < acquire a bed >;
        < set the time to end this patient's hospital stay >;
      end
  end;
procedure end_hospital_stay; { B2 }
  begin
    < discharge this patient >;
    < return the bed >;
  end;
```

Figure 2.4 Simple hospital system described in pseudo-code using three-phase approach

Since we are using Pascal, the pseudo-code used in this book is written and indented very much like Pascal, using Pascal reserved words such as *begin*, *end* and *while*. Phrases and sentences that are *not* Pascal are delimited by angled brackets.

Figures 2.4 and 2.5 show the pseudo-code for the simple hospital simulation using the three-phase approach and the event method. Appendices 2.A and 2.B show the pseudo-code for the other case studies.

2.10 SIMPLE HOSPITAL SYSTEM

Table 2.1 shows queue lengths and the use of resources in the simple hospital system, simulated over seven days where the number of *beds* is only four and the arrival times of *patients* and the lengths of stay in hospital have been predetermined rather than sampled from a distribution.

Table 2.2 shows the resulting queue lengths and the use of resources in the system. Events can be seen to occur every day except day three.

Conditional events can be seen to arise always as a direct result of, and in the same time beat as, scheduled events. In the simple hospital system, for example, if a *bed* is available following the occurrence of the scheduled event, *patient_arrival*, the conditional event *start_hospital_stay*

```
procedure patient_arrival; { B1 }
  begin
    if < a bed is available > then
      begin
        < acquire a bed >;
        < set the time for the end of this patient's hospital stay >;
      end
    else
      begin
        < put the patient in Q1 to await admission >;
      end;
    < create the next patient >;
    < set the time for this next patient to arrive >;
  end;
procedure end_hospital_stay; { B2 }
  begin
    < discharge this patient >;
    if < Q1 is not empty > then
      begin
        < admit the next patient from Q1 >;
        < set the time for the end of this patient's hospital stay >;
      end
    else
      begin
        < return the bed >;
      end
  end;
```

Figure 2.5 Simple hospital system described in pseudo-code using event method

ılso takes place. If, on the other hand, a *patient* is in the admission queue ɔecause a *bed* is not available prior to the event *end_hospital_stay*, this :vent will release a bed and cause *start_hospital_stay* to take place.

2.10.1 Three-phase approach

The pseudo-code in Fig. 2.4 shows two bound events and one conditional :vent. The bound events are numbered B1 and B2 and the conditional :vent, C1. The *patient_arrival* feeder, B1, is a very simple event which puts he arriving entity on the queue, creates a new entity and sets the time for it o arrive (i.e., the time for the feeder to be recalled). The first arrival is :reated in the initialization phase of the simulation after which each arrival ;enerates the next.

The conditional events are run independently of the bound events

Table 2.1 Patients needing admission to four-bed ward over seven-day period, showing day arrived for admission and length of stay

Patient number	*Arrival day*	*Length of stay*
1	0	4
2	1	3
3	1	4
4	1	3
5	2	1
6	4	4
7	4	2
8	4	1
9	4	2
10	5	1
11	5	1
12	6	1
13	8	3

Table 2.2 Patients, with arrival days and lengths of stay as shown in Table 2.1, progressing through simple health system; number of beds in ward is four

Day	*Arrival*	*Waiting list, Q1*	*In-patients*	*Discharges*
0	1	–	1	–
1	2, 3, 4	–	1, 2, 3, 4	–
2	5	5	1, 2, 3, 4	–
3	–	5	1, 2, 3, 4	–
4	6, 7, 8, 9	8, 9	5, 6, 3, 7	1, 2, 4
5	10, 11	10, 11	8, 6, 9, 7	3, 5
6	12	11, 12	10, 6, 9, 11	7, 8
7	–	–	12, 6	9, 10, 11
8	13	–	13	6, 12

and, therefore, may have several entities waiting for several available resources. The *while* loop ensures that as many entities as possible will be satisfied. In C1 the entities are *patients* waiting for *beds*. The hospital stay is ended by B2 and a bed is released for other waiting patients.

Table 2.3 shows, in detail, what happens on day 6 in the simulation using the three-phase approach and the data in Table 2.1.

(*a*) At the beginning of the day, the calendar contains five entities. The first one is engaged to the feeder (B1) which is due to generate patient 12 on day 6 and the other four are engaged to the activity *hospital_stay*. The time on the entity clock gives the day

Table 2.3 Three-phase simulation of day 6 of simple hospital system example, showing state of calendar and waiting list after each event; entities in calendar are identified by their next event, attribute number, and time of next bound event

Phase	*Event*	*Calendar*	*Waiting list Q1*	*Patients using a bed*
A	–	[1, 12, 6] [2, 7, 6] [2, 8, 6] [2, 9, 7] [2, 6, 8]	10, 11	7, 8, 9, 6
B	B1	[2, 7, 6] [2, 8, 6] [2, 9, 7] [2, 6, 8] [1, 13, 8]	10, 11, 12	7, 8, 9, 6
	B2	[2, 8, 6] [2, 9, 7] [2, 6, 8] [1, 13, 8]	10, 11, 12	8, 9, 6
	B2	[2, 9, 7] [2, 6, 8] [1, 13, 8]	10, 11, 12	9, 6
C	C1	[2, 9, 7] [2, 10, 7] [2, 11, 7] [2, 6, 8] [1, 13, 8]	12	10, 11, 9, 6

of discharge (i.e., the time of *end_hospital_stay*, which is B2). Patients 7 and 8 are due to be discharged on day 6 and the other two on days 7 and 8 respectively. Three bound events are therefore due to take place on day 6, the day in question.

(*b*) *B phase*. The first event, B1, generates patient 12 who is added to the waiting list, Q1, and creates patient 13 who is put in the calendar to be generated on day 8. B2 is called twice to discharge patients 7 and 8 respectively and hence release two beds.

(*c*) *C phase*. The executive now performs the C phase and tests the one conditional event, *start_hospital_stay* (C1), to see if any patients are waiting and any beds free. Patients 10 and 11 are admitted, acquiring two beds while patient 12 remains in the waiting list.

This example demonstrates two important characteristics of the three-phase approach:

(*a*) conditional events are performed separately and independently of bound events,

(*b*) entities are put in queues for conditional events in the B phase regardless of the availability of resources.

2.10.2 Event method

There are two scheduled events only shown in Fig. 2.5. The work of the conditional events has been absorbed in these scheduled events.

The feeder is now more complex because it not only generates patients, but checks whether a bed is available for each. It has an *if...then* loop for this condition, rather than a *while* loop (such as in C1 in the previous example), because only one patient is generated and tested at a time.

If a bed is not available on arrival, then a patient will acquire a bed when one is released in B2. Here again, an *if* statement is used because only one bed is released at a time. The conditions which lead to an arriving patient starting a hospital stay thus occur in more than one event.

Table 2.4 shows the progression of events on day 6 of the hospital simulation, using the event method and the data in Table 2.1.

(*a*) *At the beginning of the day* the simulation is in exactly the same state as for the three-phase simulation with five entities in the calendar and two in the waiting list.

(*b*) *B phase*. Three patient entities have day 6 on their clocks as follows.

(i) Patient 12 is taken off the top of the calendar and put in the waiting list in the feeder event, B1, and patient 13 is added to the bottom of the calendar. If, however, there had been a bed free at this stage, Patient 12 would have acquired a bed within this event.

(ii) Patient 7 is taken off the calendar and discharged in B2. Instead of releasing the bed, however, the simulation allocates the bed to the next patient on the waiting list, patient 10. This patient is then slotted into the calendar, in time order position, with day 7 as the discharge day.

(iii) Patient 8 is taken off the calendar and discharged in B2. Patient 11, the next on the waiting list, is allocated a bed and put in the calendar with discharge day 7.

***Table 2.4** Event-based simulation of day 6 of simple hospital system example, showing state of calendar and waiting list after each event; entities in calendar are identified by their next event, attribute number, and time of next scheduled event*

Phase	*Event*	*Calendar*	*Waiting list* Q1	*Patients using* a bed
A	–	[1, 12, 6] [2, 7, 6] [2, 8, 6] [2, 9, 7] [2, 6, 8]	10, 11	7, 8, 9, 6
B	B1	[2, 7, 6] [2, 8, 6] [2, 9, 7] [2, 6, 8] [1, 13, 8]	10, 11, 12	7, 8, 9, 6
	B2	[2, 8, 6] [2, 9, 7] [2, 10, 7] [2, 6, 8] [1, 13, 8]	11, 12	10, 8, 9, 6
	B2	[2, 9, 7] [2, 10, 7] [2, 11, 7] [2, 6, 8] [1, 13, 8]	12	10, 11, 9, 6

At the end of the day the calendar and lists are in exactly the same state as they were in the three-phase approach, shown in Table 2.3.

This example illustrates the greater complexity of the events in the two-phase method, where resources are allocated within the scheduled events as, and when, they become available.

2.10.3 Comparison of two approaches

The procedures of the three-phase approach are short and simple. In the event method, on the other hand, the programmer has to determine which conditional events can take place within each scheduled event. If, moreover, the simulation conditions are changed, the programmer has to determine the effects of this on all the scheduled events.

The three-phase approach is therefore more modular, easier to program, and much more robust to changes than the event method. However, the executive has to do more work because all the conditional events have to be tested after each bound event has been performed. Therefore, the three-phase approach is more efficient in development time but less efficient in computer time than the event-based method.

A further important difference is that the order in which the conditional events are listed in the calendar determines their relative priorities. In the event method these have to be identified explicitly within the scheduled events. The problems arising with modelling priorities will be discussed further in Chapter 7.

Example

(8) In the hospital system described in Appendix 2.A, the two conditional events are *start_hospital_stay* (C1) and *start_pre_operative_stay* (C2). Both of these events describe admissions to hospital wards but in C1 they come from a list of patients who do not require operations and in C2 they come from a list of patients who do require operations. When a bed becomes available and there are patients in both queues, C1 will be performed first and therefore patients who do not require operations are always given priority over the use of the beds. If those who require operations were to be given priority, then *start_pre_operative_stay* would have to be listed first in the executive.

The advantages and disadvantages of the three world views will be discussed further in Chapter 12.

2.11 PREPARING A SIMULATION MODEL

In any modelling exercise it is important first to decide the boundaries of the system to be modelled and the system objectives (briefly discussed in Chapter 1). The objectives and choice of technique will then determine the assumptions to be made (and vice versa). A much more detailed description of the stages of modelling is given by Rivett (1980). In using discrete-event simulation, it is usual to make fairly restrictive assumptions about queue discipline, and the number and the type of activities and entities. These may then be relaxed at a later stage when the data has been collected (see Chapter 4) or after model validation (see Chapter 8). Chapter 6 gives a more detailed discussion of the initial stages of preparing a simulation model.

In devising the simulation structure, the first step is to identify the entities, the activities, and the queues and then to draw an activity flow or cycle diagram.

The next step will depend both on the simulation world view and on the computer package to be used. An interactive package such as Inter_SIM (O'Keefe 1987) or a programmer generator such as CAPS (Clementson 1978) may be used immediately to generate a working simulation program. However, these have their disadvantages which will be discussed in Chapter 14. If a simulation is to be written in a tailor-made simulation language or in a general high-level language such as FORTRAN or Pascal, then the next step is to write the program structure in pseudo-code. The program is then ready for coding.

2.12 SUMMARY

Time, in a simulation model advances from one discrete point in time, called a *time beat*, to the next. Discrete-event simulations are concerned with the activities of individual *entities* which alternate from *busy* to *idle* or *queueing*, and back again, as the simulation progresses. The change of entity state is called an *event*. When entities are busy they are said to be engaged to an *activity*.

Entities generally need some *resources* in order to take part in an activity. Events which mark the start of an activity are often dependent on some condition being met, such as a resource being available. These are called *conditional events*. *Bound events* which mark the end of an activity are unconditionally scheduled to occur at a particular point in time. *Feeders* are a special type of bound event which generate new temporary entities. Activity diagrams show the flow or life cycle of entities and pseudo-code is used to describe activities in structured English.

There are three world views for controlling the structure of, and the flow of time in a discrete-event simulation: the three-phase approach, the event method, and the process view. The main difference between the three-phase approach and the event method lies in the treatment of conditional events. In the *three-phase* approach, the conditional events are treated as separate procedures, whereas in the *event* method they are integrated in the scheduled events.

2.13 EXERCISES

1. Tables 2.3 and 2.4 show the changing content of the calendar and queues in day six of the simple hospital system, using the data in Tables 2.1 and 2.2. Do the same for both the three-phase approach and the event method for days four and five.

2. A garage chain wants to determine the staffing levels in a new type of garage which will operate as follows. A receptionist registers each vehicle, passes it to a team of senior mechanics who diagnose the faults and then send it to either a minor or major repair workshop. The workshop mechanics repair the vehicle and send it to a test station. If the examiners at the test station find fault with the vehicle, they return it to the relevant workshop for correction. When they pass it, they send it to a valeting department where it is thoroughly cleaned inside and out. The cleaning staff then drive it to the reception area from where the customer can collect it.

   (*a*) List the resources, entities, and activities in this system and then draw an activity flow diagram of it.
   (*b*) Identify the decision variables.
   (*c*) Suggest what additional information you would need in order to be able to model this system.

3. The ante-natal clinic is on Wednesday afternoon. Women who attend, have appointments to see either the community midwife or the general practicioner (G.P.) but each must first visit the practice nurse to be weighed. The nurse, who has no appointment system, sees other people with various ailments as well as the pregnant women.

   New patients always see the G.P. alone. After the first visit, and until she is 35 weeks pregnant, a patient is booked to come once every four weeks and see, on alternate occasions, the G.P. and midwife. After this, and until the baby is born, she is booked to come every week and see the G.P. and midwife together. Pregnancy usually lasts 38–40 weeks. Patients needing to see both the midwife and G.P. are booked for joint appointments at the end of the clinic.

   The doctor is concerned because of complaints about long queues in the clinic and his wife is annoyed that he gets home late on Wednesdays. He asks you to look at the system to see if it could be organized more efficiently. You decide to write a simulation model.

   (*a*) Identify the elements of the system you will model. Suggest alternative ways of organizing the clinic that might improve matters.
   (*b*) Identify the entities, attributes, resources, and queues of this system.
   (*c*) Draw an activity flow chart and, based on the three-phase simulation method, write a brief description of each activity in pseudo-code.

4. *Bank System* (Appendix 1.A.1). Assume that four tills remain open all the time.

   (*a*) Draw activity flow charts for the present and proposed queuing system.
   (*b*) Write pseudo-code for the bound and conditional events in the three-phase approach.
   (*c*) Write pseudo-code for the scheduled events in the event-scheduling approach.

   State any additional assumptions that you need to make.

5. *Traffic Light System* (Appendix 1.A.2). Carry out the following tasks.
   (*a*) Draw an activity flow chart for the system.
   (*b*) Write pseudo-code for the bound and conditional events in the three-phase approach.
   (*c*) Write pseudo-code for the scheduled events in the event-scheduling approach.

   State the assumptions you need to make.

* * *

2.A CASE STUDIES IN PSEUDO-CODE USING THREE-PHASE APPROACH

The numbers in braces to the right of each procedure name indicate the number of the event. The queue numbers match those shown in the activity diagrams in the chapter.

2.A.1 Hospital system

patient1 entities represent those patients who do not have an operation during their stay whereas *patient2* entities have operations. The events of the hospital system using the three-phase approach are shown below.

```
procedure patient1_arrival; { B1 }
  begin
    < put the newly arrived patient in Q1 for admission >;
    < create the next patient >;
    < set the time of arrival of this next patient >;
  end;
procedure patient2_arrival; { B2 }
  begin
    < put the newly arrived patient in Q2 for admission >;
    < create the next patient >;
    < set the time of arrival of this next patient >;
  end;
procedure start_hospital_stay; { C1 }
  begin
    while < Q1 is not empty and a bed is available > do
      begin
        < admit the first patient from Q1>;
        < acquire a bed >;
        < set the time to end this patient's hospital stay >;
      end
  end;
```



```
procedure end_hospital_stay; { B3 }
  begin
    < discharge this patient >;
    < return the bed >;
  end;
procedure start_pre_operative_stay; { C2 }
  begin
    while < Q2 is not empty and a bed is available > do
      begin
        < admit the first patient from Q2 >;
        < acquire a bed >;
        < set the time to end this patient's pre-operative stay >;
      end
  end;
procedure end_pre_operative_stay; { B4 }
  begin
    < put patient on the end of Q3 for an operation >;
  end;
procedure start_operation; { C3 }
  begin
    while < Q3 is not empty and a theatre is open and available > do
      begin
        < make the theatre unavailable for any other patients >;
        < operate on the next patient in Q3 >;
        < set the time for the end of the operation >;
      end
  end;
procedure end_operation; { B5 }
  begin
    < make the theatre available >;
    < put the patient in Q4 to end the post-operative stay >;
  end;
procedure start_post_operative_stay; { C4 }
  begin
    < while Q4 is not empty >;
    < set the time for discharge from hospital >;
  end;
procedure end_post_operative_stay; { B6 }
  begin
    < discharge this patient >;
    < return the bed >;
  end;
procedure open_theatre; { B7 }
  begin
    < open the theatre >;
    < set the time for theatre to close >;
  end;
```

```
procedure close_theatre; { B8 }
  begin
    < close the theatre >;
    < set the time for the theatre to open >;
  end;
```

2.A.2 Repair shop system

```
procedure end_working; { B1 }
  begin
    < put the broken machine in Q1 for the removal of parts >
  end;
procedure start_removal; { C1 }
  begin
    while < Q1 is not empty and a mechanic is available > do
      begin
        < take the next machine from Q1 >;
        < acquire a mechanic >;
        < set the time for the end of the removal of
          the parts from this machine >;
      end;
  end;
procedure end_removal; { B2 }
  begin
    < put the machine in Q2 for repair >;
  end;
procedure start_repair; { C2 }
  begin
    while < Q2 is not empty and the equipment is available > do
      begin
        < take the next machine from Q2 for repair >;
        < acquire the equipment >;
        < set the time for the end of the repair to this machine >;
      end;
  end;
procedure end_repair; { B3 }
  begin
    < return the mechanic >;
    < return the equipment >;
    < set the time for the machine to stop working again >;
  end;
```

2.B CASE STUDIES IN PSEUDO-CODE USING EVENT METHOD

The two case studies are described using the event method.

2.B.1 Hospital system

```
procedure patient1_arrival; { B1 }
  begin
    if < there is a bed available > then
      begin
        < acquire the bed >;
        < set the time for the end this patient's hospital stay >;
      end
    else
      begin
        < put this patient in Q1 for admission >;
      end;
    < create the next patient >;
    < set the time for the arrival of this next patient >;
  end;
procedure patient2_arrival; { B2 }
  begin
    if < there is a bed available > then
      begin
        < acquire the bed >;
        < set the time for end of this patient's pre-operative stay >;
      end
    else
      begin
        < put this patient in Q2 for admission >;
      end;
    < create the next patient >;
    < set the time of the arrival of this next patient >;
  end;
procedure end_hospital_stay; { B3 }
  begin
    < discharge this patient >;
    if < Q1 is not empty > then
      begin
        < admit the next patient from Q1 >;
        < set the time of the end of this patient's hospital stay >;
      end
    else
      if < Q2 is not empty > then
        begin
          < admit the next patient from Q2 >;
          < set the time of the end of this patient's hospital stay >;
        end
      else
        begin
          < return the bed >;
        end;
  end;
```

```
procedure end_pre_operative_stay; { B4 }
  begin
    if < theatre is open and available > then
      begin
        < make theatre unavailable for other patients >;
        < set the time for the operation to finish >;
      end
    else
      begin
        < put the patient in Q3 to await an operation >;
      end
  end;
procedure end_operation; { B5 }
  begin
    if < Q3 is not empty and the theatre is open > then
      begin
        < operate on the next patient from Q3 >;
        < set the time for the end of the operation >;
      end
    else
      begin
        < make the theatre available >;
      end;
    < set the time for the end of this patient's post operative stay >;
  end;
procedure end_post_operative_stay; { B6 }
  begin
    < discharge this patient >;
    if < Q1 is not empty > then
      begin
        < admit the next patient from Q1 >;
        < set the time for the end of this patient's hospital stay >;
      end
    else
      if < Q2 is not empty > then
      begin
        < admit the next patient from Q2 >;
        < set the time of the end of this patient's hospital stay >;
      end
    else
      begin
        < return the bed >;
      end
  end;
procedure open_theatre; { B7 }
  begin
    < make the theatre open >;
    if < theatre is available and Q3 is not empty > then
```



```
      {Note: in theory the theatre could be unavailable because patients can
             finish their operation after it has been closed and so the program
             logic allows for this}
      begin
        < operate on the next patient in Q3 >;
        < make the theatre unavailable >;
        < set the time for the end of the operation >;
      end;
    < set the time for the theatre to close >;
  end;
procedure close_theatre; { B8 }
  begin
    < close the theatre >;
    < set the time for the theatre to open >;
  end;
```

2.B.2 Repair shop system

```
procedure end_working; { B1 }
  begin
    if < there is a mechanic available > then
      begin
        < acquire a mechanic >
        < set the time for the end of the removal of machine's broken
          parts >;
      end
    else
      begin
        < put the machine in Q1 to await a mechanic >
      end
  end;
procedure end_removal; { B2 }
  begin
    if < there is equipment available > then
      begin
        < acquire the equipment >;
        < set the time for the end the repair to this machine >;
      end
    else
      begin
        < put the machine in Q2 for repair >;
      end
  end;
procedure end_repair; { B3 }
  begin
    < set the time at which machine will stop working again >;
```

```
    if < Q2 is not empty > then
      begin
        < take the next machine from Q2 >;
        < set the time for the end of the repair to this machine >;
      end
    else
      begin
        < return the equipment >;
        if < Q1 is not empty > then
          begin
            < take the next machine from Q1 >;
            < set the time for the end of the removal of the machine's
              broken parts >;
          end
      end
    else
      begin
        < return the mechanic >;
      end
end;
```

3 Representing a Simulation in Pascal

Once the simulation logic has been determined, it must be represented as a computer program. There are numerous purpose-built programming languages and packages designed for this task. Three of the most popular are SIMSCRIPT (Russell 1983), GPSS (Schriber 1974), and SIMULA (Birtwistle *et al.* 1979). The major advantage of using a simulation programming language (SPL) is that it imposes structure on the model. This structure will be based on one of the world views mentioned in the previous chapter. Most SPLs also provide comprehensive facilities for sampling, list control, entity descriptions, the collection of results, and report writing.

An alternative to using an SPL is to use a general purpose high-level language such as Pascal or FORTRAN. Those using this approach are well-advised to use a set of pre-written routines that provide the basic facilities of an SPL. This will ensure that they write their simulation programs in an efficient and reliable way that is adaptable to changes in the model structure and which can be readily understood by others in the field.

This book uses a Pascal package, Pascal_SIM, in which most of the commonly used facilities are provided. Readers can translate these concepts and contructs to another high-level language if they prefer.

Pascal_SIM which can be used for the three-phase approach, the event method and the process view is based on the following concepts:

(*a*) entities carry the information about their next bound (or scheduled) events and the time at which they will occur,
(*b*) queues and the calendar are implemented as lists; entities thus move from one list to another as they progress through the simulation,
(*c*) the executive is programmed as a procedure which must be updated and recompiled for each new simulation model.

This chapter shows how the simulation logic can be coded in Pascal using the variables, procedures, and functions available in Pascal_SIM. Pascal_SIM is documented and shown in detail in the appendices at the back of the book.

3.1 PASCAL

In choosing a high-level language in which to implement simulation facilities, we have sought the following characteristics:

(*a*) good list-processing facilities,
(*b*) a language that is widely used and encourages good programming practices, and
(*c*) portability.

Pascal appeared to fulfil the criteria.

Pascal was designed as a structured programming language in the early 1970s by Nicklaus Wirth and has become widely used in many educational institutions, now being the main scientific high-level language taught to students.

There is an internationally recognized standard for Pascal, the ISO standard (Wilson and Addyman 1982), which many implementations of Pascal follow. A program developed using the ISO standard is thus easily transferable from one machine to another. With a few exceptions, the ISO standard is followed in this book. (Appendix A itemizes these exceptions; Appendix C shows how to implement Pascal_SIM with a number of Pascal implementations.)

The facilities of Pascal which make it particularly appropriate for simulation work are those providing a record structure for variables, for typing, and for linking records. These are described in more detail below.

3.1.1 Record structure

Variables may be declared to have a record structure with fields of particular types (e.g., *integer, character, boolean*). This is extremely useful for a simulation program where entities, resources and queues all have particular characteristics that need to be represented.

3.1.2 Typing

Pascal is 'strongly typed'. Every variable must be declared with a specific type, which determines how it can be used in a program. Predefined types are *integer*, *real*, *char* and *boolean*. Many Pascal implementations also provide a *string* type although this is not in the ISO standard.

In Pascal it is also possible for users to define their own types. For example a *scalar*-type declaration that is used in Pascal_SIM is:

cardinal = 0..*maxint*;

The advantage of defining **cardinal** in the range of 0 to *maxint* (*maxint* being the largest integer variable a particular computer can store) is that positive variables can be declared as type **cardinal** rather than *integer*, enabling the program to perform range checking. Any person reading the program will also know that the program requires a positive integer in the specified range.

There are also *enumerated* types. For example, Pascal_SIM includes a type for screen color thus:

color = (**null**, **black**, **red**, **green**, **yellow**, **blue**, **magenta**, **cyan**, **white**);

A variable which is declared as type **color** can then take the value of any of the colors in the list.

The facility for typing makes programs both more concise and more readable.

3.1.3 Pointers

Pointer variables are variables which locate other variables. Variable *A* is said to 'point at' variable *B* if *B* can be located by dereferencing *A*. Therefore, pointer variable *B* does not need a unique name. Pointer variables, in general, do not need unique names because they can be located via other variables.

Another important characteristic is that computer memory space for a pointer variable is not created when it is declared but explicitly created in a program using the procedure *new*. This memory allocation may be disposed of using the procedure *dispose*. This is important in simulation programs, where the number of entities in the system may vary considerably over the duration of the program.

Record pointers are used extensively in the procedures described in this book and, therefore, it is desirable that the reader should understand what they do and how they work. Readers unfamiliar with pointers are recommended to refer to the book by Findlay and Watt (1985).

The linking facility by which records can be dynamically created and disposed of will be shown to provide an ideal facility for manipulating queues and the calendar list.

3.2 ENTITY TYPE

3.2.1 Entity attributes

The last chapter explained that entities have an attribute for holding the time of the next bound event, called the entity clock. The simulation also

needs to know which bound event they are to take part in next. They thus need: attributes which enable them to be individually identified, their entity clocks, and attributes to indicate which bound events they will take part in next. The simulation may also need to know whether they are busy or idle in order to determine whether they are available to take part in certain activities. Furthermore entities belong to classes which distinguish between groups of entities. In Pascal_SIM, the basic entity attributes are as follows:

(*a*) an indication of availability (a Boolean) which is set to *true* if an entity is available and *false* if it is taking part in an activity and is thus in the calendar,
(*b*) a class number,
(*c*) a color attribute for visual simulations (see Chapters 9 and 10 – this is not covered here),
(*d*) an attribute number to uniquely identify individual entities,
(*e*) the next bound event (denoted by a number), and
(*f*) the time of the next bound event.

An entity record in Pascal_SIM (omitting the color attribute) thus has the following structure:

```
type
  an_entity = packed record
                avail          :boolean;
                attr, next_B   :cardinal;
                class          :class_num;
                time           :real;
              end;
```

where

(*a*) **cardinal** = 0..*maxint*;
(*b*) **class_num** = 1..**max_class_num**; and **max_class_num** is set to the maximum number of classes.

An entity is then defined as a pointer variable:

```
type
  entity = ^an_entity;
```

In a simulation model, entities can be declared as type **entity**. For example, a patient entity in the hospital example can be declared as follows:

```
var
  patient :entity;
```


However, entities do not normally have unique variable names because they are stored in lists.

3.2.2 Creation and disposal

When entities are created they acquire computer memory space and when disposed of, they release it again. Pascal_SIM supports temporary entities and thus entities can be created or destroyed at any point in the simulation program. Entities can be created with a class and attribute number by using the following function:

```
function new_entity (c :class_num; a :cardinal) :entity;
```

For example,

```
var
  patient :entity;
patient := new_entity (1,1);
```

creates a new patient entity with class number 1 and attribute 1. The class number indicates the group of items to which an entity belongs and the attribute number can be used to uniquely identify each entity. Entities which leave the system can be removed using:

```
procedure dis_entity (e :entity);
```

3.2.3 Current entity

Although, as mentioned above, the different entities in a simulation are not normally given unique names, they are given temporary names when they are in use. When an entity is taken off the calendar to be used in a bound event it is referred to as the current entity and called **current**. For example, the statement:

```
dis_entity (current);
```

disposes of the current entity, regardless of its class or attribute number.

3.3 RESOURCE TYPE

Resources are passive and must be acquired from and returned to *bins*. Bins are record variables representing a set of resources of the same type;

```
type
  bin = record
          number,num_avail :cardinal;
        end;
```

where

(*a*) **number** is the total number of resources of that type allocated, and
(*b*) **num_avail** is the number available at any point in time.

For example, the beds in a ward are represented by a **bin** in the hospital examples as follows:

```
var
  bed :bin;
```

In order to set the maximum number of resources available in any particular **bin**, the procedure **make_bin** is called. For example:

make_bin (*bed*, 10)

establishes a **bin** with 10 beds (i.e., there are 10 beds in the ward). The total number of beds is thus **bed.num**, and the number available is **bed.num_avail**. The following procedures:

procedure **acquire** (*var from* :**bin**; *n* :**cardinal**);
procedure **return** (*var from* :**bin**; *n* :**cardinal**);

are used to acquire or return a number of resources, denoted by *n*, to the **bin**. A resource may be required for several activities. It is acquired when it is used at the beginning of an activity and returned at the end of the last activity in which it is needed. An attempt to acquire a resource from a bin that contains no available resources, or return a resource to a bin when all resources are available, results in an error message.

3.4 LISTS

Pascal_SIM uses a double-linked circular chain to represent a list. From each link in the chain, a pointer can be moved forward to the next link, or backward to the previous link.

The chain is composed of pointer variables called **links**. The structure of a **link** type is as follows:

```
type
  link   = ^a_link;
  a_link = record
```



```
             next,pre :link;
             item     :entity;
           end;
```

Link points to an entity, denoted as **item**, and also points to a previous and a next link each of which also points to an entity. Each **link** thus has no characteristics of its own but exists to relate one entity to another in an ordered way. If a list contains no entities, one link remains, in order to keep the list open and available. This remaining link is called the list 'head' and points to a dummy entity. This is shown in Fig. 3.1.

Figure 3.2 shows that when an entity is added to the empty list, a new link is created whose previous and next link are both pointing at the head link. At the same time, the pointers of the head link are also changed so that its previous and next links are the new link.

In general, when an entity is added to a list a new link is created in the closed chain. Figure 3.3 shows that the links on either side of the new link have to stop pointing to each other and have to point to the new link. Figure 3.4 shows that the opposite happens when an entity is removed from a list.

Lists are used to represent queues, the calendar, and class lists. These are described in more detail below.

3.4.1 Queues

A queue is defined to be a **link** type:

```
type
  queue = link;
```

A **queue** may be created as an empty list using:

procedure **make_queue** (*var q* :**queue**);

This creates the list head, and points the head's **pre** and **next** pointers at the head.

In a queue, the entities are ordered from 'top' to 'tail'. The entity at the 'top' of the list is defined to be the one next to the head and the 'tail' is previous to the head. These programming simulations will usually want to put entities on, or take entities off, either the top or tail of a queue. The following procedures and functions provide these facilities:

procedure **give_top** (*q* :**queue**; *e* :**entity**);
procedure **give_tail** (*q* :**queue**; *e* :**entity**);
function **take_top** (*q* :**queue**) :**entity**;
function **take_tail** (*q* :**queue**) :**entity**;

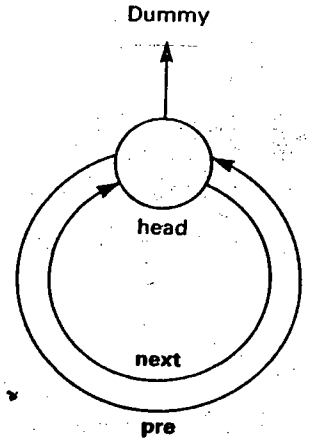

Figure 3.1 Empty list with head's next and pre links pointing to itself

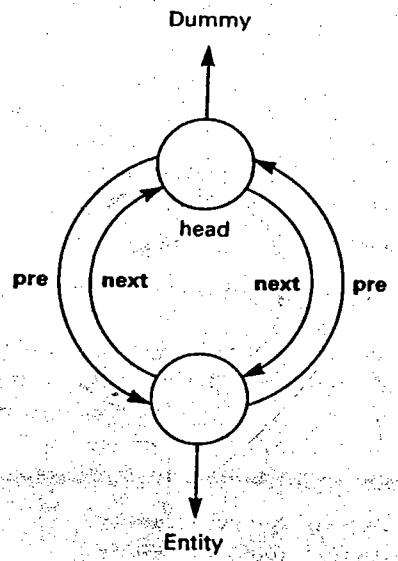

Figure 3.2 List with one entity; head links point to link with entity and vice versa

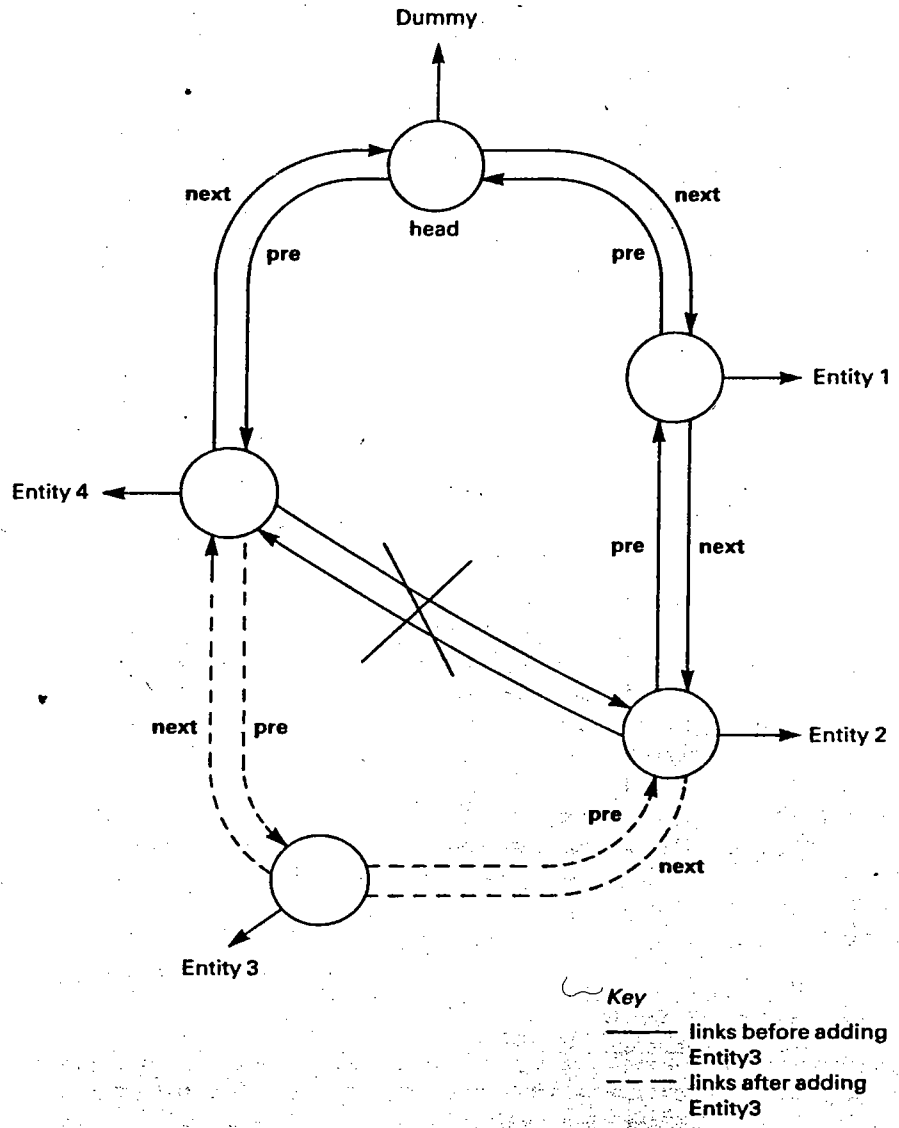

Figure 3.3 Addition of entity to list; entity 3 is added to list between Entities 2 and 4

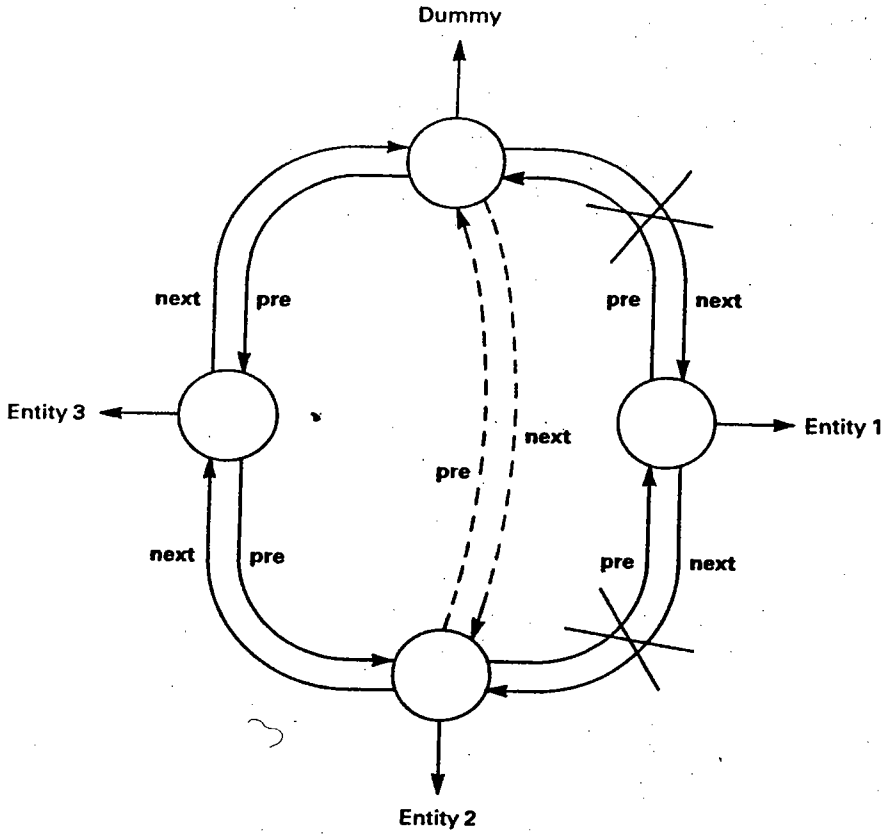

Figure 3.4 Removal of entity from list; entity 1 is removed from top of list and entity 2 replaces it

For example,

```
var
  patient_q :queue;
give_tail (patient_q, current);
```

will put the **current** entity on the end of a queue called *patient_q* and

```
patient := take_top (patient_q);
```

takes the entity off the top of *patient_q*, which now becomes the entity *patient*. Entities may be added or removed from *any place* in a queue using:

```
procedure give (q :queue; t :link; e :entity);
```

and

```
function take (q :queue; t :link);
```


In using these functions, the links after which an entity is to be added or at which an entity is to be removed have to be specified explicitly. The entity to be inserted is always the **next** one after the one specified by *t* in **give**.

In the last chapter, we described how entities queue for resources before taking part in activities. These queues are constantly changing in size and composition and it is frequently necessary to check the length of the queue. The function which does this is:

```
function count (var q :queue) :cardinal;
```

This function will count all the entities in the queue. The function **empty** checks whether a queue is empty or not:

```
function empty (q :queue) :boolean;
```

returning *true* if *q* does not contain any entities, otherwise returning *false*.

3.4.2 Calendar

Chapter 2 defined the calendar to be the list of entities which are scheduled to take part in bound events. This is implemented in Pascal_SIM as a queue, called **calendar**, which is ordered according to the value on the entity clock, **time**. The next entity due to take part in an event (i.e., the one with the smallest value of **time**) is kept at the top of the **calendar** (i.e., is on the next link to the head) and the last one due to take part in an event (i.e., has the largest value of **time**) is at the tail. When an entity is added to the list, it is put in after an entity with a smaller or identical value of **time** and before an entity with a larger value of **time**. Entities are always removed from the top of the calendar.

3.4.3 Class lists

It is often useful to search through entities of the same class. This can be done efficiently if they are linked together in a queue. This *class list* is maintained independently of either the queues for activities or the calendar. Instead of creating entities as individuals with **make_entity**, entities may be created and linked in a class using:

```
procedure make_class (var c :queue; n,size :cardinal);
```

For example,

```
var
  pat :queue;
make_class (pat, 1, 30)
```

creates 30 entities numbered 1–30 in a queue or class called *pat* with class number 1.

Class lists are usually used for permanent entities which are always kept fixed in number. However, all the procedures and functions for adding entities to queues and removing them from queues can also be used for class lists.

3.5 TIME ADVANCE AND EXECUTIVE

Simulation time is advanced in discrete steps from one time beat to the next. The current simulation time Pascal_SIM, is a global variable, **tim**, which must never be altered by the programmer.

3.5.1 Cause statement

Activity times for entities are usually scheduled within conditional events or feeders (see Chapter 2). The procedure **cause** schedules the entities. It calculates the time on the entity clock by adding **tim** to the scheduled activity duration, sets the entity clock and the number of the next event and enters the entity in the calendar in time order. It is:

procedure **cause** (*nb* :**cardinal**; *e* :**entity**; *t* :*real*);

where:

(*a*) *nb* is the unique number given to each bound event,
(*b*) *e* is the entity, and
(*c*) *t* is the duration of the activity.

For example,

```
var
  q1 :queue;
cause (2, take_top(q1), 10);
```

takes the entity from the top of the queue called q1 and puts it in the calendar in time order so that it does bound event number 2 in 10 time units from the current simulation time. The entity time of entity *e* is thus set to **tim** + 10, where **tim** is the present simulation time and **next_b** is set to 2.

3.5.2 Executive

The executive is contained in the following procedure:

procedure **run** (*duration* :*real*; *max_C* :**cardinal**);

where

(*a*) *duration* is the duration of the simulation,
(*b*) *max_c* is the number of conditional events in the simulation. (This is set to zero in a two-phase simulation using the event method.)

Figure 3.5 shows the skeleton of a three-phase executive program for Pascal_SIM. The bound event numbers and names are in a *case* statement and the conditional event numbers and names are in a *for* loop.

It should be noted that the name listed for each event in the executive must correspond exactly to the name of the corresponding procedure. As the procedure **run** calls the event procedures by name, this procedure has to be adjusted and recompiled for each simulation.

There is a global Boolean variable called **running** which must be set to *true* for the executive to call and perform events (i.e., for the simulation to continue to run). If at any point in the simulation, **running** is set equal to *false*, the simulation will stop at the end of that time beat. **Running** is always set equal to *false* when:

(*a*) there are no more events remaining in the calendar, or
(*b*) the clock time on the next entity in the calendar (i.e., the time of the next event to be performed) is greater than the value of *duration*.

The three phases of the executive in Pascal_SIM are thus as follows:

(*a*) *A phase*. Assuming that **running** is true, the executive advances the simulation time by setting the current simulation time, **tim**, to the clock time on the entity at the top of the calendar.
(*b*) *B phase*. The executive performs all the bound events due at that simulation time; they will be performed in the order in which they happen to have arrived on the **calendar**.
(*c*) *C phase*. The executive tests the conditional events, in the listed order, to see whether any of their conditions have been satisfied to enable them to be performed. They will, in their turn, put more entities into the calendar and thus keep the simulation running.

The executive for the event method has exactly the same structure but without the C phase.

3.6 SIMULATION STRUCTURE

In order to run a simulation there must be initialization and reporting facilities as well as the events and the executive.

```
      procedure run (duration :real; max_C :cardinal);
      { duration is the duration of the simulation and
        max_C is the number of C events }
        begin
          < set Boolean variable running to true >
          repeat
            if < calendar is empty > then < set running to false >
              else
                begin
                < set the simulation time to the time on the entity at
                  the top of the calendar >
A phase         if < present simulation time has exceeded the duration >
                    then < set running to false >
                  else
                    begin
                      while < there is an entity in the calendar and the time
                              of the entity at the top of the calendar is equal
                              to the simulation time > do
                        begin
B phase                   < take entity off the top of the calendar >
                          case < the number of the B event on the entity > of
                            0 :
                            1 :   < first B event >;
                            2 :   < second B event >;
                              :
                              :
                          end;
                        end { do };
                      for < number of C event, from 1 to the max_C > do
                        case < number of C event > of
                          1 :   < first C event >
C phase                   2 :   < second C event >
                              :
                              :
                        end { for };
                    end
                end
            until not running;
          end { run };
```

Figure 3.5 Executive of three-phase approach in pseudo-code showing three phases

The procedure **initialize** is written by the user and sets up the initial queues and classes using:

(*a*) **make_sim** (to set up the calendar),
(*b*) **make_class** (to define the classes),
(*c*) **make_queue** (to set up the queues),

(*d*) **make_bin** (to set up the resources in bins),
(*e*) **make_streams** (to set up the streams for sampling, see Chapter 4),
(*f*) **make_histogram** (to set up the histograms for collecting statistics, see Chapter 5).

It is essential that the initialization phase creates at least one entity, using **new_entity**, and puts it in the calendar with the **cause** statement to start the time advance in the executive.

The **report** procedure is for the presentation of statistics at the end of a simulation run (see Chapter 5). The simulation program structure is then:

```
bound events;
conditional events;
procedure run;
procedure initialize;
procedure report;
  begin
    |
    initialize;
    run;
    report
    |
  end.
```

3.7 SIMPLE HOSPITAL SYSTEM

Appendix 3.A shows a three-phase program of the simple hospital simulation which uses the facilities of Pascal_SIM, listed in Appendix B at the back of the book. (In order to get Pascal_SIM and this program running, readers are advised to look at the implementation details in Appendix C.) The structure of the program follows the pseudo-code shown in Fig. 2.4.

In this simulation there are 20 hospital beds. The patient interarrival times and the hospital stay are fixed at constants 6 and 60 time units, respectively. The simulation is set to run for 720 time units. The program writes out the arrival and departure time of each patient. Note that **report** is a dummy procedure because there is no collection of statistics.

3.8 SUMMARY

This chapter has shown how the simulation concepts described in Chapter 2 can be implemented in Pascal_SIM. Pascal_SIM provides *entities* which can be individually identified and belong to *classes*. They can be created as

they are required and disposed of when no longer needed. *Resources* are available which can be *acquired* and *returned* to their bins as necessary.

Linked lists are the most important and useful structures as they keep ordered sets of entities. These form the basis of *queues*, the *calendar*, and *class lists*.

The simulation executive is very similar both for the three-phase approach and the event method. The executive controls the time advance and access to the events. The overall structure of the simulation program must include initialization, events, reporting facilities and the executive.

Chapter 6 explains and shows the full simulation programs for the hospital and machine repair case studies, including the initialization and report phases.

* * *

3.9 EXERCISES

In order to implement Pascal_SIM for use in these exercises, you can either copy the entire code from Appendix B and implement it on your machine using the instructions in Appendix C, or you can implement just those parts you need. For these exercises you will need the following routines and corresponding global variables:

(*a*) error messages,
(*b*) queue processing,
(*c*) entities and classes,
(*d*) resources, and
(*e*) the three-phase and event executive (i.e., procedure **run**).

Appendix D shows a full set of routines for non-visual simulations.

1. Write Pascal_SIM statements to do each of the following:
   (*a*) create the 30th entity in a class of customers (class number three) and call it *customer*;
   (*b*) while there is an entity in the queue called *wait_list* and a resource is available called *hire_car*, acquire the resource and dispose of the entity at the top of the queue;
   (*c*) create a class list of 20 entities in a class called *lorry* (class number two);
   (*d*) search the queue *wait_list* for an entity with attribute equal to six;
   (*e*) cause the current entity to perform bound event number two in 12.1 time units from the present simulation time, **tim**;
   (*f*) perform the executive of a three-phase simulation with 19 conditional events for 100 time units;
   (*g*) perform the executive of the model in (*f*) using the event method.
2. Using Pascal_SIM, write the executive of a launderette which is to run for 50 time units and has the following events: *customer_arrives* (B1), *start_wash* (C1), *end_wash* (B2), *start_dry* (C2), *end_dry* (B3).

3. Compile and run the simple hospital system using the program in Appendix 3.A. Print out the time of each event to check that the simulation is working properly.
4. *Bank System.* Model the present queueing system in the bank (Appendix 1.A.1) in Pascal_SIM using the pseudo-code developed in Exercise 4 of Chapter 2. Build up either a three-phase or two-phase simulation in the following stages:
   (*a*) outline the structure of the simulation;
   (*b*) write the executive;
   (*c*) declare the entities, resources, queues and variables;
   (*d*) write the events assuming that the customers arrive at one minute intervals and each spends exactly three minutes being served;
   (*e*) write procedure **initialize**;
   (*f*) put the parts together into a program and compile;
   (*g*) provide output to check that the simulation is working logically.
5. *Traffic Light System.* Using the traffic light system described at the end of Chapter 1 (Appendix 1.A.2) and the pseudo-code developed in Exercise 5 of Chapter 2, write a working program in Pascal_SIM in the same way as in the previous exercise. Assume that the cars arrive at half-second intervals.

* * *

3.A SIMPLE HOSPITAL SYSTEM CODED IN PASCAL_SIM

```
program simulation (output);

  const
    amount_of_beds      = 20;
    inter_arrival_time = 6;
    stay_time           = 60;

  var
    bed :bin;
    q   :queue;

{ Bound events }

procedure patient_arrives;  { B1 }
  begin
    writeln ('Patient arrives at time:  ',tim:7:2);
    give_tail (q,current);
    cause (1,new_entity(1,1),inter_arrival_time);
  end;
```

```
procedure end_hospital_stay;  { B2 }
  begin
    writeln ('Patient departs at time:   ',tim:7:2);
    return (bed,1);
    dis_entity (current);
  end;

{ Conditional events }

procedure start_hospital_stay; { C1 }
  begin
    while (bed.num_avail>0) and (not empty(q)) do
      begin
        writeln ('Patient admitted at time: ',tim:7:2);
        acquire (bed,1);
        cause (2,take_top(q),stay_time);
      end;
  end;

procedure run (duration :real; max_C :cardinal);
  var
    c :cardinal;
  begin
    running := true;
    repeat
      if calendar=calendar^.next then running:=false
        else
          begin
            tim := calendar^.next^.item^.time;
            if duration<tim then running := false
              else
                begin
                  while (calendar<>calendar^.next) and
                    (tim=calendar^.next^.item^.time)  do
                    begin
                      calendar_top;
                      case current^.next_B of
                        0: ;
                        1: patient_arrives;
                        2: end_hospital_stay;
                      end;
                    end;
                  for c:=1 to max_C do
                    case c of
                      1: start_hospital_stay;
                    end;
                end;
          end
    until not running;
  end { run };
```



```
procedure initialize;
  begin
    make_sim;
    make_streams;
    make_bin (bed,amount_of_beds);
    make_queue (q);
  end { initialize };

procedure report;
  begin
  end;

begin
  initialize;
  cause (1,new_entity(1,1),0);
  run (720,4);
  report;
end.
```

4

Sampling from Distributions

This chapter describes how to generate random numbers and to sample from frequency histograms and common parametric distributions. It also shows some of the difficulties and pitfalls of collecting and analyzing data for use in a simulation program.

4.1 NEED FOR SAMPLING

In management and organizational systems, the underlying processes will usually be stochastic in nature. In queueing systems, for example, the average arrival and service times can be measured but it is not possible to predict when people or items will arrive, which queue they will join, and how long it will take for them to be served. As a simulation has to describe individual arrivals and activities, it must, if it is to demonstrate the stochastic nature of a system, sample different interarrival and service times and decisions. Moreover, the frequency distribution of the samples should look like those measured in the real system. Distribution sampling techniques are thus needed.

A sample from a uniform distribution is equally likely to be any number in a specified range. Although this is very simple, other distributions such as the negative exponential, the Normal and log Normal are often much more useful for describing the distributions of arrival and service times. If it is not possible to fit the data to a common parametric distribution, then it may be necessary to sample from a relative-frequency histogram of the observations.

Distribution sampling uses sequences of random numbers. Whereas one simulation run may be adequate to observe the general behavior of a system, several runs with activity times and decisions based on different sequences of random numbers, are likely to be needed to provide adequate statistical measures for comparative or predictive purposes.

4.2 RANDOM NUMBER GENERATION

A number is said to be *random* if its occurrence is unpredictable from any previously sampled numbers and is equally likely to occur anywhere over a predefined range. Random number generators usually provide real numbers in the range (0, 1).

In a sequence of random numbers, each must be independent of *all* previous ones. Any sequence generated from a *formula*, by definition, produces predictable and therefore non-random numbers. Truly random numbers must be generated from random phenomena of the type which occur in radio-active or electronic gadgets. These are necessary for some simulations where complete randomness is of the utmost importance, but they have the serious disadvantage that, unless they are recorded on file, they cannot be recalled and re-used for multiple runs of a simulation program. As very many numbers are invariably needed, random-number storage is unsatisfactory and wasteful of storage space.

Most simulation packages therefore use *pseudo-random numbers* which can be generated from formulae and *appear* to be unpredictable and independent of each other although, of course, they are not. There has been much research into devising formulae which can generate streams of numbers which have properties close to those of genuine random numbers.

4.3 PSEUDO-RANDOM NUMBERS

A common method of generating pseudo-random numbers is based on the production of a sequence of integers over a wide range, using the following simple iterative process:

$$n_{i+1} = f(n_i)$$

where the number, n_{i+1}, appears to be unrelated to n_i and to as many previous numbers in the sequence as possible.

In order to produce real numbers in the range (0, 1), the integers are divided by the width of the range of integers, m. The pseudo-random number u_i is thus found as follows:

$$u_i = \frac{n_i}{m}$$

As m is large, the resultant sequence of real numbers, $\{u_i\}$, will appear to have been sampled from a continuous distribution. In a good generator, each number will also appear to have come, with equal probability, from anywhere in the interval.

All generators of this type will *cycle*. This happens when the genera-

tor returns to a number that has occurred before. For example, if n_i is equal to n_j, then $f(n_i)$ is equal to $f(n_j)$ and therefore n_{i+1} is equal to n_{j+1}. Hence, all subsequent numbers in the sequence will also recur in the same order.

The recycling of numbers is clearly non-random and provides external evidence of a dependence between the numbers. It is important in divising a pseudo-random number generator, therefore, to make the cycle length as long as possible so that the generator produces very many different integers before it recycles.

A useful class of functions is given by the formula:

$$f(n_i) = (an_i + c) \bmod m$$

where a, c, and m are integer constants. Thus $f(n_i)$ is the *remainder* when a linear transformation of the number, n_i, is divided by m. The next integer in the sequence, n_{i+1}, is set equal to $f(n_i)$. The values of the generated numbers are sometimes large and sometimes small and, depending on the value of the constants, can appear to vary almost unpredictably.

If the value of c is greater than zero then the generator is called a *mixed congruential generator*. The formula will generate integer numbers in the range 0 to $m - 1$. It is clearly important, therefore, that m should be a very large number.

The maximum cycle length of this type of generator is thus m but it may be smaller than this if some integers in the range are never produced. For example, if all the chosen constants are even, then no odd numbers are generated. The rules to ensure that the cycle length is maximal are quite complicated and were devised by Knuth (1969):

(a) c and m are relatively prime (i.e., they have no common factors),
(b) if p is a prime factor of m then choose $a = 1 \pmod p$,
(c) if 4 is a factor of m then choose $a = 1 \pmod 4$.

Note that if condition (c) holds then condition (b) also holds.

If c is equal to 0, then the generator is a *multiplicative congruential generator* and only one rule is necessary: that m and a should be mutually prime. In this case, however, the number 0 is never generated and the maximum cycle length is $m - 1$ rather than m.

A further principle for either type of generator is the avoidance of small multipliers (constant a) because they cause the appearance of sequences of monotonically increasing numbers within the number stream.

Many generators which obey these rules nevertheless have inherent and detectable patterns. They must be subjected to a battery of statistical tests to ensure that numbers are equally likely to occur in any range and that they are independent. There is considerable research in this field covering many different types of generator. Examples include Fishman and More (1982), IBM (1969) and Lewis *et al.* (1969). These tests are not

discussed in this book and for a comprehensive coverage, the interested reader should read the work by Fishman (1978), or Law and Kelton (1982).

Pascal_SIM has generators of this type. These are as follows:

16-bit: $f(n_i) = (3993n_i + 1) \bmod 32\,767$
32-bit: $f(n_i) = 16\,807n_i + 0) \bmod 2147\,483\,647$

The 32-bit generator, which is well established is a reliable generator, was recommended by Lewis *et al.* (1969). The 16-bit generator, which is useful for 16-bit microcomputers, relies on the non-detection of integer overflow (Thensen *et al.* 1984). It is inevitably inferior but, nevertheless, it is adequate for development work and investigative simulations.

4.4 STREAMS

A pseudo-random number generator starting on the same number or *seed* will always produce the same stream of numbers. Different seeds will give rise to different streams and if the seeds do not happen to be numbers which are close together in the cycle, the generated numbers will appear to be independent of each other.

When comparing the effects of different policies on the outcome of simulation runs, it is important that a comparison run should use the same activity times and decisions as the original run except where these are directly affected by the predetermined change in input. It is thus necessary to re-use the same random numbers for different policy options. Chapter 8 will show that to reduce the variance of the results, the length of time taken for each activity and the decision taken at each branch in a simulation should be sampled from different random number streams. Therefore, there should be sufficient streams for each activity and each decision branch. Moreover, when several simulation runs are used to find the means and variances of results, each run should be based on different and independent random number streams.

In Pascal_SIM, the procedure **make_streams** initializes all 32 streams by setting the values of the seeds of each stream. Seed i is given the value $1000\,i + 7$ to provide adequate spacing in the 16-bit generator. The spacing should, ideally, be increased for the 32-bit generator.

The array **original_seeds** holds these initial seed values and the array **seeds** holds the current integer random number values in each stream. Thus, whereas **make_streams** initializes the values of all the streams, an *individual* stream j can be reset independently of the others by

```
seeds [j] := original_seeds [j]
```

The following function is used to sample a number between 0 and 1 for any of the 32 streams:

function **rnd** (*s* :**stream_num**) :*real*;

Each time a program calls **rnd**, with any specific stream number, the next number in the sequence is produced by the pseudo-random number generator and **seed** is updated.

4.5 SAMPLING FROM DISCRETE DISTRIBUTIONS

A discrete probability distribution is the distribution of probability over a number of discrete points:

$$x_1, x_2, x_3, x_4, \ldots, \quad \text{where } x_{i+1} > x_i$$

In simulation problems, the points will usually be finite in number and occur at integer points: 0, 1, 2 . . . m. If the probability that i occurs is given by p_i, then

$$\sum_{i=1}^{m} p_i = 1, \quad \text{over all values of } i$$

Figure 4.1 shows an example in which a function is defined at points 0, 1, 2, 3, 4, 5 with probabilities: ⅛, ⅛, ¼, ¼, 0, ¼, respectively.

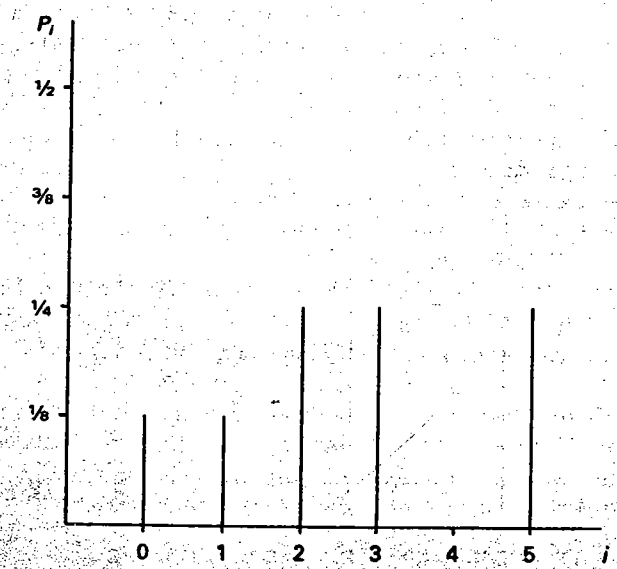

Figure 4.1. Discrete probability function defined at points 0–5

4.5.1 General discrete distributions

Simulations use discrete distributions when:

(*a*) there is a branch in a simulation model such that the choice of branch for each entity is random and unrelated to any previous activities or entity characteristics (see Chapter 7);
(*b*) newly generated entities have to be given characteristics (see Chapter 7), which may be chosen from a discrete distribution. Patient entities, for example, may be given an age group and blood group.

Suppose a finite discrete distribution has m points with a probability, p_i, at each point i. In order to decide which of the m points to choose, the simulation generates a pseudo-random number between 0 and 1 from one of the streams. The decision is based on the range of numbers within which the random number falls. For m paths, it is done in the following way:

Range of numbers	*Choice of path*
$0, p_1$	1
$p_1, p_1 + p_2$	2
$p_1 + p_2, p_1 + p_2 + p_3$	3
. . .	. . .
. . .	. . .
$\sum_{i=1}^{k-1} p_i, \sum_{i=1}^{k} p_i$	k
. . .	. . .
. . .	. . .
$\sum_{i=1}^{m-1} p_i, 1$	m

The width of each range of random numbers is thus equal to the probability. These ranges can be seen to be based on the cumulative probability function:

$$F(k) = \sum_{i=1}^{k} p_i$$

Using this notation, the path k is chosen if the random number lies between $F(k-1)$ and $F(k)$.

The general method of sampling a value, x_i, from a discrete distribution with probability function, $P(x)$, and a cumulative probability function, $F(x)$, in pseudo-code is as follows:

< sample a random number u in the range (0, 1) >;
< set i to 1 >
while < u is less than $F(x_i)$ > *do* < increase i by 1 > ;
< x_i is sampled from the discrete distribution >

In Pascal_SIM, the discrete values, x_i, and their corresponding cumulative distribution values $F(x_i)$, can be held in a text file, referred to as **sample_file**. **Make_sample**, which is called at the beginning of a simulation run, reads these values into a two-dimensional array called *table*. The first column in the table has the cumulative values: $F(x_1)$, $F(x_2)$ ... $F(x_n)$ and the second column consists of: x_1, x_2, ..., x_n. Using the logic described above, the probabilities of x_i are sampled using:

function **sample** (*table* :**lookup_table**; *s* :**stream_num**);

where **lookup_table** is a two-dimensional array type.

4.5.2 Poisson distribution

If the number of arrivals in any time interval are completely random and independent of those in any other time intervals, then the probability of a particular number of arrivals in any time interval (with the same average arrival rate) may be described by the Poisson distribution.

This is clearly a very useful distribution but, nevertheless, it is not commonly used in discrete-event simulations. This is because discrete-event simulations usually generate arrivals using interarrival-time distributions (such as the negative exponential distribution) rather than generating the numbers of arrivals per time interval. However, a Poisson distribution *can* model the sizes of *batches* of items arriving in the system, where the batch sizes vary from one to the next.

The method of sampling from the Poisson distribution is similar to the method for the general distribution but, in this case, the cumulative values can be calculated as required rather than stored. If the mean number of arrivals is λ, then the probability of x arrivals is given by:

$$f(x) = \frac{\lambda^x \exp(-\lambda)}{x!}, \quad 0 < x < \infty, \quad \lambda \text{ is the distribution mean}$$

For any particular distribution, the values may be calculated iteratively:

$$f(x+1) = \frac{\lambda^{x+1} \exp(-\lambda)}{(x+1)!} = \frac{\lambda f(x)}{x+1}, \quad \text{where } f(0) = \exp(-\lambda)$$

The cumulative values may also be calculated iteratively:

$$F(x+1) = F(x) + f(x+1), \quad \text{where } F(0) = f(0) = \exp(-\lambda)$$

The pseudo-code of the algorithm for the Poisson distribution is thus as follows:

< set x to 0 >
< set P, the probability of x=0, to exp(−m) >
< set F, the cumulative function, to P >
< sample a number u in the range (0, 1) >
while < u is less than F > *do*
begin
< increase x by 1 >;
< set the new value of P to (m/x)P >;
< set the new value of F to F+P >;
end
< x is the sampled value from the Poisson >
end;

The Pascal_SIM function is:

function **poisson** (*m* :*real*; *s* :**stream_num**) :*real*;

where *m* is λ the distribution mean.

4.6 SAMPLING FROM CONTINUOUS PARAMETRIC DISTRIBUTIONS

If we wanted to sample a real number, such as the length of time of an activity, this would be sampled from a continuous distribution. The *probability density function* $f(x)$ describes the distribution of the probability of x over a continuous range.

If $f(x)$ is continuous and defined between the limits a and b, then if x_1 and x_2 lie between a and b, the probability of sampling a number between x_1 and x_2, is given by:

$$\int_{x_1}^{x_2} f(x)\,dx, \quad \text{where } \int_a^b f(x)\,dx = 1$$

The cumulative probability at point y is the probability of sampling a point less than or equal to y and is expressed as a function:

$$F(y) = \int_a^y f(x)\,dx$$

The *inverse transform method* of sampling exploits the fact that $F(y)$ ranges between 0 and 1 as x goes from a to b.

Figure 4.2 shows that if we sample a random number, u, between 0 and 1, for any value of u, there is a corresponding value of $F(x)$. Thus if

$u = F(x)$, and the inverse exists, then
$x = F^{-1}(u)$

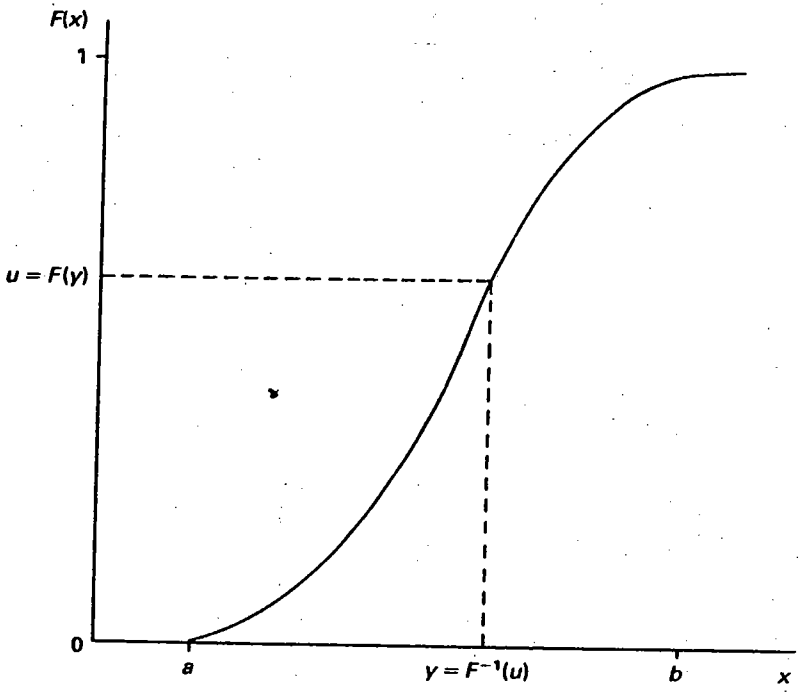

Figure 4.2 Cumulative probability function of continuous distribution, f(x), showing that, using inverse transform method, a random number, u, sampled between 0 and 1 provides y, a sample from probability distribution f(x)

To prove that x is a sample from $f(x)$, we have to show that the probability that u is sampled between two random numbers u_1 and u_2, is the same as the probability that x lies between x_1 and x_2 where

$$x_1 = F^{-1}(u_2) \quad \text{and} \quad x_2 = F^{-1}(u_2)$$

The probability that x lies between x_1 and x_2 is

$$\begin{aligned}\int_{x_1}^{x_2} f(x)\,dx &= \int_{a}^{x_2} f(x)\,dx - \int_{a}^{x_1} f(x)\,dx \\ &= F(x_2) - F(x_1) \\ &= u_2 - u_1\end{aligned}$$

which is the probability that u is sampled between u_1 and u_2.

This sampling method is very useful for probability functions for which $F(x)$ can be evaluated and has an inverse.

4.6.1 Uniform

Although the uniform distribution does not commonly occur in practice, it is a very simple distribution and useful for setting up a first draft of a simulation program before more accurate data are collected.

The uniform distribution is given by the formula:

$$f(x) = \frac{1}{(b-a)}, \quad \text{where } a \leqslant x \leqslant b$$

Since each random number u is generated from a uniform distribution, where $a = 0$ and $b = 1$, then a sample from a general uniform distribution can be derived from a simple linear transformation:

$$x = a + (b - a)u$$

In Pascal_SIM, this is provided by:

function **uniform** (*l*,*h* :*real*; *s* :**stream_num**) :*real*;

where l and h are the lower and upper limits of the range, respectively.

4.6.2 Negative exponential

Arrivals in a system are commonly assumed to be randomly distributed, independent of each other, and equally likely to occur in any time period. The number arriving in each time interval thus has a Poisson distribution, and the time *between* arrivals has a negative exponential distribution.

The negative exponential distribution is also sometimes used to describe the time taken for an activity to take place. There is an underlying assumption, however, that the time taken to finish an activity is independent of the time it has already taken. If, for example, in the machine repair problem described in Chapter 1, the probability of a machine breakdown were assumed to be independent of the time since the last repair, then the length of time between the last repair and the machine breakdown would be taken to be a negative exponential distribution.

Because this assumption is unlikely to be true in most systems, the negative exponential distribution is much more useful for interarrival times than service times.

Samples from a negative exponential distribution are obtained using the inverse transform method. The probability function is given by:

$$f(t) = \lambda \exp(-\lambda t), \quad t > 0$$

then

$$F(x) = \int_0^x \lambda \exp(-\lambda t)\,dt = 1 - \exp(-\lambda x)$$

$F(x)$ is continuous and strictly increasing and so for each value of x there is unique value of $F(x)$ between 0 and 1.
If $F(x) = u$ where u is a random number between 0 and 1, then

$$x = F^{-1}(u)$$

Therefore

$$\begin{aligned} u &= 1 - \exp(-\lambda x) \\ 1 - u &= \exp(-\lambda x) \\ \log_e(1-u) &= -\lambda x \\ x &= \frac{-\log_e(1-u)}{\lambda} \end{aligned}$$

Let $v = 1 - u$, then v is also a random number in the range 0–1, and

$$x = \frac{-\log_e v}{\lambda}$$

To obtain a sample from a negative exponential distribution, therefore, we must sample a number v from the uniform distribution and transform it using this formula. This is done in Pascal_SIM with:

function **negexp** (*m* :*real*; *s* :**stream_num**) :*real*;

where m is the distribution mean which is equal to $1/\lambda$.

4.6.3 Normal

Activity times which vary randomly about an average value may usefully be approximated by a Normal distribution. Typical activities of this type include the time it takes for someone to drink a pint of beer, for an athelete to run a race, or for a mechanic to mend a machine with a particular well-defined fault.

The Normal distribution can never be a true representation of activity time because it is unbounded whereas activity times have a minimum value of zero. As a rule of thumb, to use the Normal distribution, the mean of the activity-time distribution should be at least three standard deviations in value. Less than one percent of the values sampled will then be less than zero. If this condition does not hold, it might be better to fit the data to a skewed distribution, such as the log Normal. Whenever the Normal distribution is used, the programmer should check for and eliminate negative values because they may arise, by chance, however, large the mean.

The cumulative standard Normal distribution function is given by the formula:

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-t^2/2)\, dt$$

As there is no analytic inverse function, $F^{-1}(u)$, a choice has to be made between other, less efficient, methods of sampling from this distribution, of which there are many. The interested reader can find detailed descriptions of these in other textbooks, for example, Law and Kelton (1982).

Pascal_SIM uses the method of Box and Muller (1958) in which a pair of random normal deviates are produced from two random numbers. In order to derive these, we start with two independent random numbers sampled from a standard Normal distribution, x and y. Their joint distribution is:

$$F(x,y)\, dx\, dy = \frac{1}{2\pi} \int_{-\infty}^{y} \int_{-\infty}^{x} \exp[-(x^2+y^2)/2]\, dx\, dy$$

Using polar co-ordinates

$$x = r\cos\theta \quad \text{and} \quad y = r\sin\theta$$

Thus

$$r^2 = x^2 + y^2 \quad \text{and} \quad \theta = \tan^{-1}(y/x)$$

The joint distribution of r and θ becomes

$$\begin{aligned} F(x,\theta) &= \frac{1}{2\pi} \int_0^{\theta} \int_0^{r} \exp(-r^2/2)\, r\, dr\, d\theta \\ &= \int_0^{\theta} \frac{1}{2\pi} d\theta \int_0^{r} \exp(-r^2/2)\, r\, dr \end{aligned}$$

This indicates that θ and r are independently distributed. Let $G(\theta)$ be the cumulative distribution of θ

$$G(\theta) = \int_0^{\theta} \frac{1}{2\pi} d\theta = \frac{\theta}{2\pi}, \quad 0 \leqslant \theta < 2\pi$$

Thus θ has a uniform distribution over $(0, 2\pi)$.
If u is a random number from the range (0, 1) then

$$u = G(\theta) = \frac{\theta}{2\pi}$$

Let $H(r)$ be the cumulative distribution of r

$$H(r) = \int_0^{r} \exp(-r^2/2)\, r\, dr$$

By substitution of $s = r^2/2$, $\frac{ds}{dr} = r$, we have

$$H(r) = \int_0^{r} \exp(-s)\, ds = [-\exp(-s)]_0^{r} = 1 - \exp(-r^2/2)$$

If v is a random number from the range (0, 1) then $w = 1 - v$ is also a random number from the range (0, 1) if

$$v = H(r) = 1 - \exp(-r^2/2)$$
$$w = \exp(-r^2/2)$$

Therefore

$$r = \sqrt{-2 \log_e w}$$

We can now express the Normal deviates x and y in terms of the random numbers u and w.

$$x = \sqrt{-2 \log_e w}\ (\cos 2\pi u)$$
$$y = \sqrt{-2 \log_e w}\ (\sin 2\pi u)$$

Simulations generally need only one sample at a time and cannot therefore readily make use of the two independent Normal deviates produced by two random numbers. In Pascal_SIM we use only the first of these two. The function is:

function **normal** (*m,sd :real*; *s* :**stream_num**) :*real*;

where *m* and *sd* are the mean and standard deviation respectively.

4.6.4 Log Normal

Service times in queueing systems can often be successfully described by the log Normal distributions. A log Normal distribution has two parameters: its mean, m, and standard deviation, s. A sample from a log Normal is the logarithm of a corresponding sample from a Normal distribution, $N(\mu, \sigma)$ whose parameters are simply related by the following formulae:

$$\mu = \log_e m - \tfrac{1}{2} \log_e (s/m + 1)$$
$$\sigma = \log_e (s/m + 1)$$

If x is sampled from a Normal distribution (with parameters μ and σ) using the Box–Muller method, then $y = \log_e x$ is a sample from a log Normal distribution with parameters m and s. The Pascal_SIM function is:

function **log_normal** (*m,sd :real*; *s* :**stream_num**) :*real*;

where m and sd are the mean and standard deviation of the distribution, respectively.

4.7 SAMPLING FROM HISTOGRAMS

If the activity times or arrival times do not readily fit a common parametric distribution, then it will be necessary to sample from a probability distribution derived from a frequency histogram of activity or arrival times. Difficulties which may be encountered in finding and analysing the data will be discussed later in this chapter.

Figure 4.3 shows a histogram representing the lengths of stay of patients in hospital in a short stay ward. The discrete probability distribution is derived from the frequency distribution and can be treated in exactly the same way as the discrete probability distributions in the previous section. Figure 4.4 shows the cumulative distribution function corresponding to the histogram in Fig. 4.3 and indicates how a random number of 0.64 gives rise to a sampled length of stay of three days.

This method, however, samples lengths of stay at fixed points in time rather than from a continuous range. Samples from a continuous range are derived from the continuous cumulative distribution function, such as the one shown in Fig. 4.5. If the random number, u, is between $F(x)$ and $F(x + 1)$, then the sampled value, s, will be between x and $x + 1$. The distance that s is from x is in proportion to the distance u is from $F(x)$. Thus

$$s = x + \frac{u - F(x)}{F(x + 1) - F(x)}$$

The pseudo-code is as follows:

```
< sample a random number u in the range (0, 1) >
< set x to 1 >
while < u is less than F(x) > do x := x + 1;
```

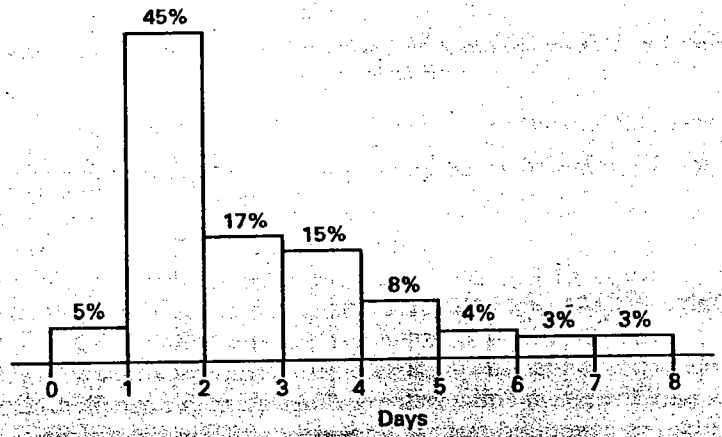

Figure 4.3 Histogram showing lengths of hospital stay in days

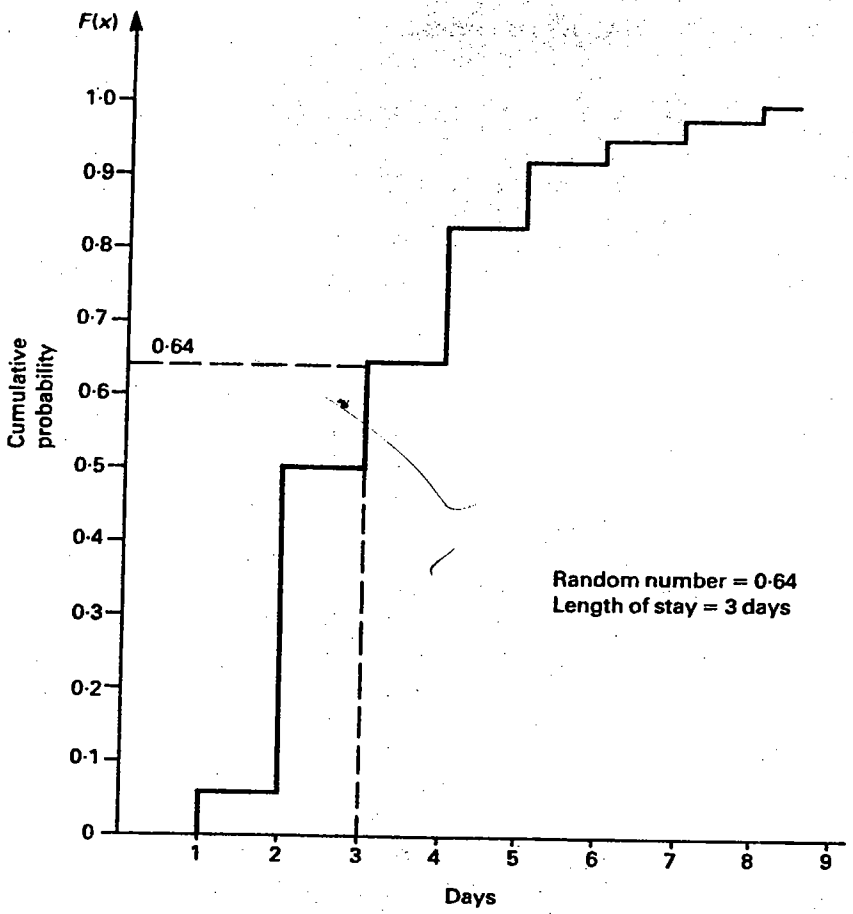

Figure 4.4 Stepwise cumulative probability function of probabilities shown in Fig. 4.3

< Set the sampled value equal to $(x - 1) + \dfrac{(u - F(x - 1))}{(F(x) - F(x - 1))}$ >

.8 CONDITIONAL DISTRIBUTIONS

ometimes simulations are initialized with entities already engaged in ctivities and their remaining activity times must be sampled. Unless a ngth of stay is generated from the negative exponential distribution, the maining activity time will be dependent on the length of time that the

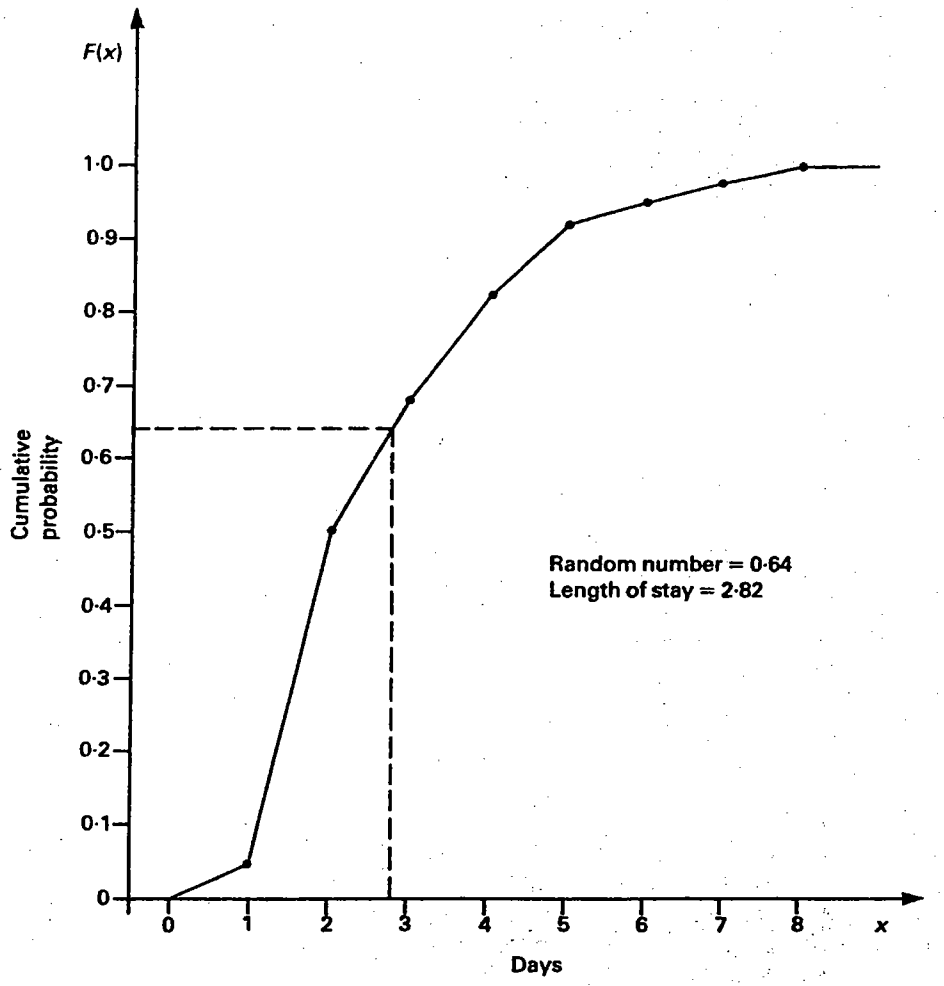

Figure 4.5 Piecewise continuous probability function of histogram in Fig. 4.3

entity has already spent on the activity. There are two approaches to sampling from the *conditional distribution*.

First, if the entity has been engaged in the activity for a period of time, T, then one samples in the usual way from the appropriate distribution to obtain, t, the time to be spent on the activity. If the value is less than T, the sample is rejected and resampled until a value, t, greater than T is found. The additional time to be spent in the activity is then given by $t - T$.

A second and more efficient method can be incorporated into the inverse transformation method. Figure 4.6 shows a cumulative function. The

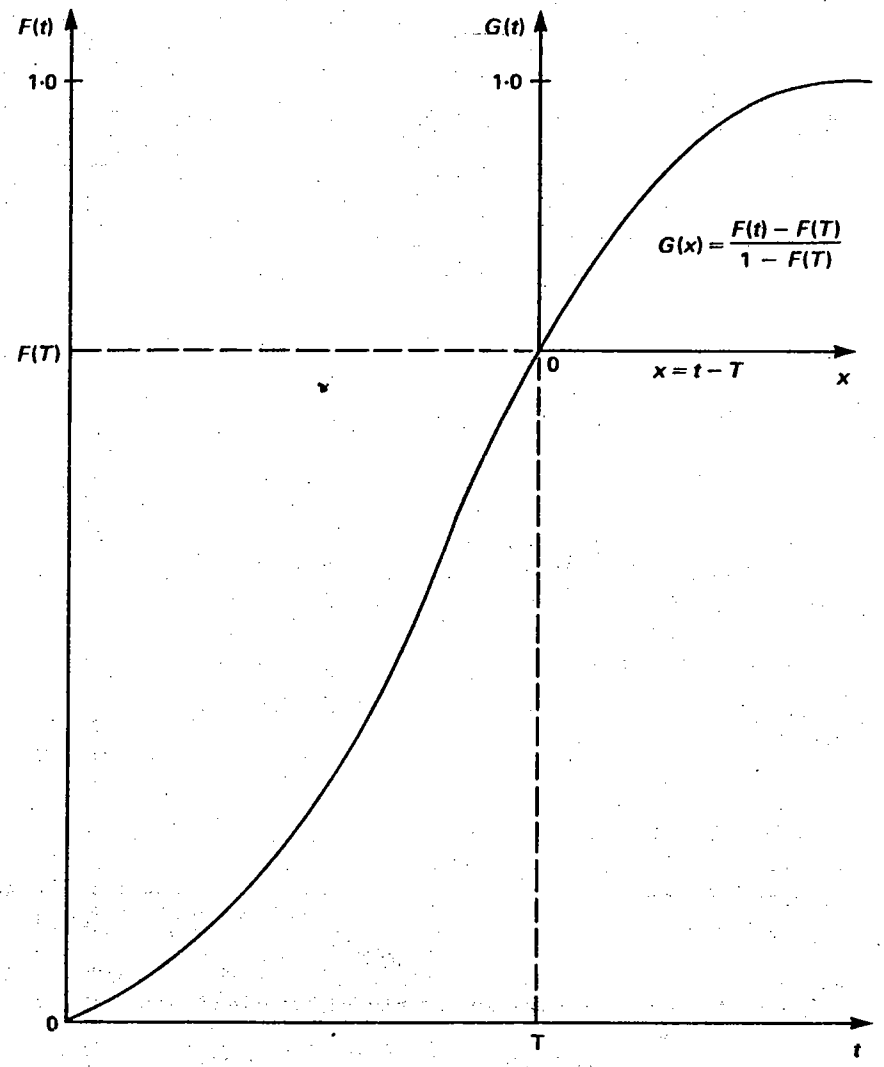

Figure 4.6 Graph showing G(x), cumulative distribution of conditional probability distribution f(t/t > T)

part of the function that is to the right of T and above $F(T)$ gives the shape of the cumulative function which is conditional upon $t > T$. However, the vertical axis must be rescaled so that the values of the cumulative probability from T to ∞ can vary between 0 and 1, rather than between $F(T)$ and 1. The new cumulative probability function to be used in the algorithm, where $x = t - T$, is thus:

$$G(x) = \frac{F(t) - F(T)}{(1 - F(T))}$$

4.9 COLLECTION OF DISTRIBUTION DATA FOR SAMPLING

Much has been written about the problems and techniques to be used in sampling distributions. PascaL_SIM together with most other reputable simulation packages provide adequate sampling facilities for most simulations. However, there are many pitfalls that arise in collecting and analysing the statistics to provide the histogram data or distribution parameters from which to sample.

It is sensible to write and test a simulation using simple parametric distributions such as the uniform or the negative exponential. Nevertheless, it is essential at an early stage in the project to ascertain the type, quantities, and reliability of data that are already available and the analyses that will be needed. Adequate data-collection exercises and surveys can then be established in good time.

There will undoubtedly be dependencies between entity characteristics, distribution data, and other factors external to the system being simulated. Considerably more data will be needed to study these than to examine the distributions and characteristics under assumptions of independence.

Decisions also have to be made between using parametric distributions and frequency histograms. With enough effort, a parametric distribution can be found to fit almost any set of distribution data. However, if a distribution fails to fit one of the common distributions described in this chapter, it will probably be quicker to sample from the frequency histogram. The penalty for this is the need for additional computer storage for the frequency data in the simulation program. Moreover, there is often a problem with the distribution tails where very little data may be available.

The Chi-squared distribution is very useful for testing whether data comes from a particular distribution but there must be sufficient data points to obey the Chi-squared rules. There should, for example, be at least five data points in each interval. For more information consult a standard statistics text such as Freud and Walpole (1980). For practical work, it is a good idea to use one of a number of commercially available distribution fitting packages such as UNIFIT (Law and Vincent 1986).

There are no simple rules for estimating the amount of data necessary to derive the shape of histograms to be used to sample activity times. However, sufficient data should be collected so that they *look* as if they approximate to a continuous distribution. Where data is thin, in the distribution tails, the values may have to be estimated by assuming a geometric decline.

Chapter 1 explained how to set objectives and plan the simulation in close co-operation with management staff. The development of the simulation structure is an interative process which is influenced by the interviews with management staff and the data analysis, as well as the later stages of simulation development such as validation.

4.9.1 Data sources

All organizations collect large quantities of data for day to day management and for accounting purposes. Time and motion studies, for example, provide extremely detailed data. Furthermore, it is quite common for the same information to be collected by different individuals or groups for different purposes.

Those needing data for simulations will need the co-operation of management in order to gain knowledge of, and access to, these information sources. They must also expect to have to do a certain amount of detective work of their own to find out exactly what is available. It may be possible to check the accuracy of the data by cross-checking information from different sources. However, this can become a time consuming red herring. If the data collection is already computerized, this can save a considerable amount of time. The main decision to be made is whether the data is good enough for the purpose or whether a new data collection exercise will have to be set in progress.

4.9.2 Data collection

Where the existing data sources are inadequate there are two main options for acquiring the data:

(*a*) mounting a special data-collection exercise, or
(*b*) acquiring parameter estimates.

Data collection can be extremely time consuming and may also present difficulties of an internal political nature because it will almost certainly be necessary to involve local staff to make the necessary observations. It is essential, therefore, to consult fully at an early stage with those who commissioned the simulation, both to agree the costs and to determine the logistics.

Parameter estimates should be used where the costs of acquiring the data outstrip the perceived benefits. The estimates should come from well-informed people in the organization and cross-checked with management staff. The sensitivity of the simulation to these estimates can be tested in the simulation, using techniques described in Chapter 8.

4.10 SUMMARY

Simulations take samples from distributions in order to provide realistic activity times and decision criteria. *Pseudo-random number generators* provide *streams* of numbers in the range (0, 1). In a good random number generator these numbers appear to be independent of each other, have no detectable pattern, and each is equally likely to appear anywhere in the range.

In the *inverse transform method*, the random number is set equal to the inverse of the cumulative distribution, to sample a number from the distribution. This principle is used in sampling from frequency histograms, the exponential distribution and the Poisson distribution. As the cumulative Normal distribution has no analytic inverse, other methods have been developed. The Box–Muller method is described in this chapter. A sample from a log Normal distribution uses a transformation of the parameters sampled from a Normal distribution.

There may be many different distributions in one simulation. Decisions about which parametric distributions to use, the parameter values, and the frequency histograms must be based on collected data and information from those who are working on, or are knowledgeable about the system being simulated.

4.11 EXERCISES

In using PascalSIM for these exercises, note that Exercises 1–6 require only the routines for random number generation and streams. The remaining exercises require these routines together with the ones listed in section 3.9. You should take particular care to read the instructions in Appendix C if you are implementing the 32-bit generator.

1. Check that your random number generator is working by:
   (*a*) printing out 50 random numbers from the first five streams, and
   (*b*) simulating 360 tosses of an unbiased six-sided die and plotting the results as a relative frequency distribution.
2. Use Pascal to simulate the raffle described in Exercise 2 of Chapter 1 and estimate the expected winnings.
3. Estimate the profit for different values of price in the following example:

   Profit = Normal (1000,100) * (Price − Normal (20,2)) − Normal (20000,5000)

4. A speculator is considering whether to buy a pair of linked investments (i.e., he can invest in neither or both). The expected pay-offs from the two investments are sampled from the following probability distributions:

(*a*) Normal with mean \$100000 and standard deviation \$70000,
(*b*) Negative exponential with mean \$150000.

respectively. Write a Monte-Carlo simulation to provide an estimate of the total payoff from the two investments and log the results to a histogram.

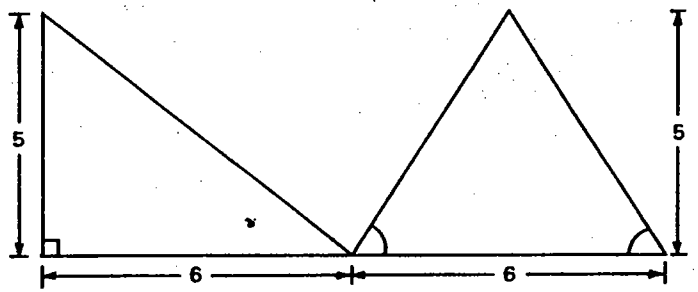

Figure 4.7 Area of shape in Exercise 5

5. Write a program to find the area of the shape shown in Fig. 4.7 using a Monte-Carlo simulation. Calculate the exact area of the shape and estimate the number of samples necessary for your simulation to provide a good approximation to it. Does the use of different random number streams affect this number?

6. Write an additional procedure for Pascal_SIM to sample from a histogram. Test the procedure on the histogram shown in Fig. 4.3.

7. Amend the simple hospital system (Chapter 3, Appendix 3.A) to sample random arrivals, with an average interarrival time of six hours, and sample the lengths of stay from a log Normal distribution with a mean of 60 hours and a standard deviation of 20 hours.

8. *Bank System*. Look at the data in the case study at the end of Chapter 1 (Appendix 1.A.1) and do the following:

   (*a*) fit a negative exponential distribution to the customer arrival data;
   (*b*) fit distributions to the overall service-time data, the Till 2 data, and the service-time data excluding Till 2;
   (*c*) substitute sampled values in the **cause** statements in the program developed in Exercise 4 of Chapter 3;
   (*d*) suggest what additional data you would like to collect from the bank in order to simulate this system satisfactorily.

9. *Traffic Light System* (Appendix 1.A.2). Carry out the following tasks:

   (*a*) fit a Poisson distribution to the vehicle-arrival data and hence derive the interarrival time distributions,
   (*b*) discuss what non-random factors may be present in the vehicle arrival data,

   (*c*) adapt the program developed in Exercise 5 of Chapter 3 to sample arrivals from the negative exponential distributions.
   (*d*) identify what other data should be collected in order to meet the objectives of the study and suggest how this should be done.

5

Collection and Analysis of Results

Data may be collected from any activity or queue in any time beat of a simulation run. There is a temptation, therefore, to produce mountains of output. Although the objectives of the study should determine the exact selection to be made, most simulation runs should produce a few digestible results which can be summarized using graphs or histograms, or in simple tables. Statistical analyses, such as factor analysis and the estimation of confidence intervals, both of which are described later in this chapter, are also helpful.

5.1 RESULTS FROM DIFFERENT TYPES OF SIMULATIONS

Chapter 1 listed three purposes of simulation modelling, each of which has different requirements for the collection of simulation results and their analysis.

(*a*) *Comparison*. Simulation runs are designed to compare the effects of changing the values of a decision variable, such as an arrival rate or a resource level. Values may be plotted and compared by eye or by the use of statistical hypothesis testing techniques. Interest will centre on an assessment of the *difference* between output measures from two or more runs.

(*b*) *Prediction*. A simulation for predictive purposes will provide averages or trends in the same way as a simulation for comparative purposes. However, it will be important to obtain absolute, rather than comparative, parameter estimates of distribution values and to evaluate changing trends over time. Several runs, each with different sets of random number streams, are likely to be needed.

(*c*) *Investigation*. A simulation set up to investigate how a system works should provide information about changing resource use and queue lengths with time. A continuous display of information while a simulation is running is particularly helpful, highlighting particularly long queues, the underuse of resources and other problems. The use of continuous displays for validation and to assess how a system works will be discussed in Chapters 9 and 10. The accuracy of statistical estimates of values is of much less importance.

The simulation objectives determine the output measures to be collected. For comparative and predictive simulations, data must be captured while the simulation is running to provide frequency distributions, estimates of distribution parameters of values, and to display trends over time. 'Snap shots' of queue values, entity numbers, or resource use at particular points in time may also be useful.

5.2 STEADY STATE AND TERMINATING SIMULATIONS

A variable is in a *steady state* if its average value remains the same over the time period under consideration. A simulation is in a steady state if all its queues are in a steady state. Most simulations will attain a steady state if they are run for a long enough period of time.

Some simulations describe systems which operate over a short period of time, such as the repair shop simulation which takes place over a single eight-hour shift. A more dramatic example would be the simulation of the launch of a rocket, which could never be expected to attain a steady state. These are called *terminating simulations*.

Figure 5.1(*a*) shows the behavior of parameters in a steady-state simulation which, when started without any active entities, attains stability after an initial *run-in* period (sometimes called the *initial transient* period). It is only of interest while it is in its steady state. A terminating simulation may also attain a steady state, but in this case the whole duration of the simulation, rather than just the steady state, is of interest. Figure 5.1(*b*) shows the typical pattern of queue behavior in shops and banks, which starts at zero, attains a steady state, and then declines again to zero at the end of the day.

Not all simulations, however, will be terminating or steady state. Queues may fail to reach a steady state in the chosen duration of the simulation. Moreover, the simulation may not have a natural terminating point. For example, Davies (1985b) describes a non-terminating simulation of patients with chronic renal failure who increase in number with increasing queues for kidney transplants, throughout the simulation run.

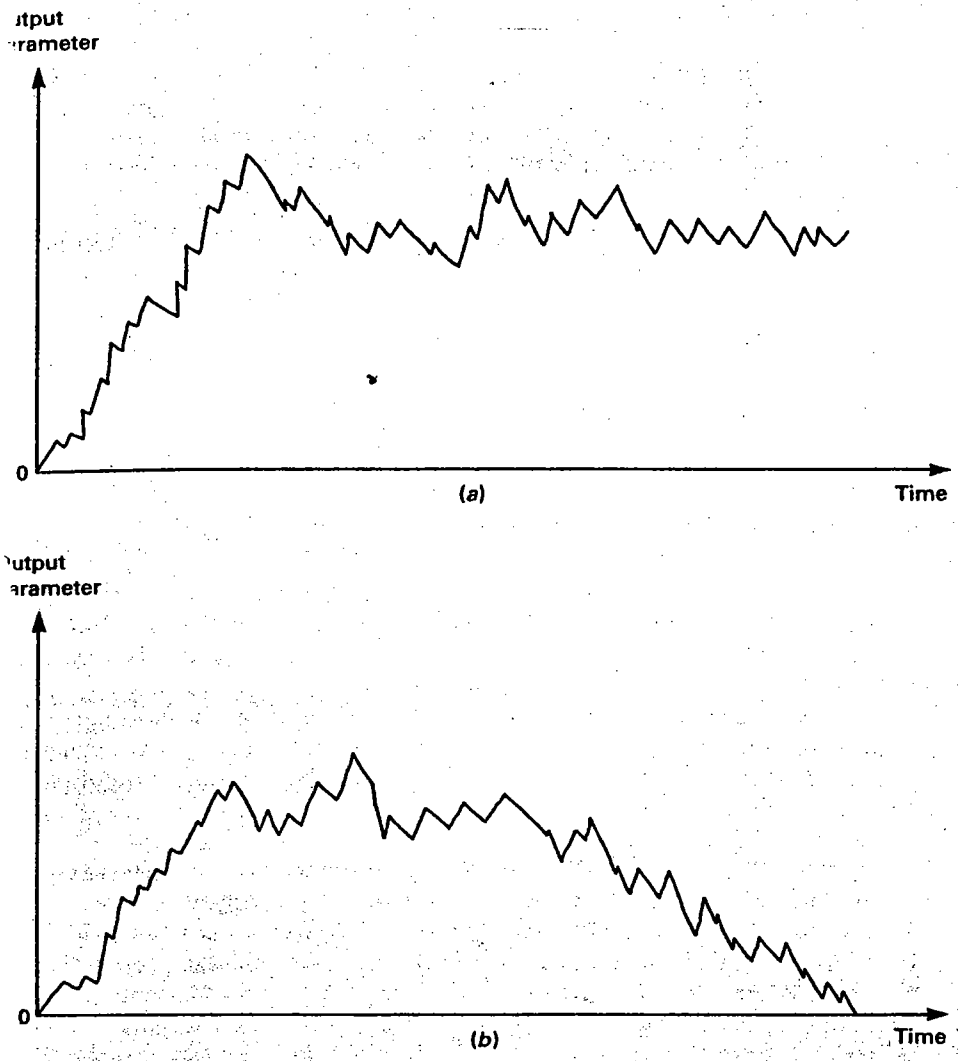

Figure 5.1 Typical behavior of parameters, such as queue length in (a) steady state, and (b) terminating simulations

5.2.1 Starting conditions and initial bias

With a steady-state simulation, the *starting conditions*, i.e., the initial values of all parameters, will influence the time taken to reach a steady state. If results are collected over the whole duration of the simulation, the run-in period will influence the aggregate results and corresponding parameter estimates. This *initial bias* can be offset, to some extent, by running the simulation for a very long period of time.

For accurate results, the simulation should be in a steady state during the whole period over which results are collected. This is done in one of three ways.

(*a*) The simulation is started in the steady state by priming it with information from the 'real-life system', including the number and types of entities in activities and queues. All the entities that start in activities must be put in the calendar and given a time at which they will end their activities. If the service time has a negative exponential distribution, then the time to end the activity can be sampled in the usual way. For other distributions, it is more accurate to sample from the conditional distribution (see Chapter 4).

(*b*) The simulation is run up to the steady state once and its finishing point used as a starting point for all subsequent runs.

(*c*) A simulation is run from empty up to the steady state. Statistics are collected from the simulation once it has reached a steady state. Any results from the run-in period are rejected.

In using any of these methods, care has to be taken to ensure that a steady state has been reached. Moreover, with the first two methods there is a danger that the simulation is *biased* by a very abnormal set of starting conditions. There is also a danger that a simulation, which starts in a steady state with one set of values for the decision variables, may not start in a steady state if those values are changed. The third approach is thus generally the most reliable.

5.2.2 Detecting steady state

Steady state can normally be adequately detected by use of the cumulative moving average method (Gafarian *et al.* 1978). Here, cumulative means for each output parameter are generated at discrete-time points, and plotted. As the system reaches a steady state, the cumulative moving average for each parameter will tend towards a constant. The point at which the constant is reached can be detected by eye, or by the use of simple statistical techniques. Since the output parameters may behave differently, the end of the run-in period is the time at which the *last* parameter reaches a steady state.

If $\bar{x}_t$ is the mean value of a parameter in time period t, then subsequent cumulative moving averages are $\bar{x}_1$, $(\bar{x}_1 + \bar{x}_2)/2$, $(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)/3$. The

time interval width must be chosen so that there are a reasonable number of observations. This can only be determined by trial and error.

In most simulation packages, including Pascal_SIM, the cumulative moving average can be calculated from statistics recorded after each chosen time interval. Chapter 6 shows how to determine the steady state for the hospital simulation.

5.3 QUEUEING THEORY AND SIMULATION

Discrete-event simulation models may be regarded as complex queueing models. The results produced from a queueing theory model are, therefore, also of interest to those using simulation models. These include:

(*a*) queue lengths,
(*b*) waiting times in queues,
(*c*) numbers of entities in the system,
(*d*) throughput time (i.e., time spent by entities in the system),
(*e*) utilization rates or resource use.

These results may be weighted and summarized to produce costs.

Systems described by queueing models are assumed to be in a steady state and therefore one or two parameters may adequately describe the results of interest. Simulations which do not describe steady-state systems are likely to be much more complex in structure than queueing models and many more results may need to be collected and presented.

Unlike queueing models, results from simulations are summarized from individual measurements recorded at the time of each event. Moreover, the measurements are dependent on the random numbers produced in any particular run. The collection of data from a simulation program is akin to the collection of data from activities in 'real life' where decisions about the type and accuracy of the required results will influence the period of time over which the results are collected. However, simulations have the advantage that repeat runs can be made, with different random number streams, to provide more accurate estimates of results.

5.4 COLLECTION OF RESULTS

The types of results that a simulation may produce are as follows.

(*a*) The distributions, means, and variances of response variables over time (such as the changing queue length and resource use) provide particularly useful information about steady state simulation runs.

(*b*) Time series plot and analyses are, on the other hand, only of interest for variables which vary systematically during part or whole of a simulation run. They are thus used in non steady state simulations.
(*c*) Single variables taken at a 'snap shot' in time (such as queue lengths and counts of entities) or results which summarize measurements over a whole run, such as total or average resource use, may be of interest in steady state or non steady state simulations.

Simulations must record the statistics in histograms or tables as the simulation progresses.

5.4.1 Waiting times of entities

The normal place to record the waiting times of entities in queues is at the beginning of a conditional event when the entity is removed from the queue, and before it is put into the calendar again.

To calculate the waiting time, the start of the wait must be recorded. Fortunately, this is done automatically in a simulation program because, if the entity has been in the calendar, the time attribute on the entity still has the value of the time when the last bound event took place; this is usually the time that that entity was put in the queue. If the entity has not been in the calendar, then it has almost certainly been in the same queue since the beginning of the simulation and the time attribute has value zero. The time spent in the queue is therefore the 'present time' minus the time on the entity clock.

5.4.2 Time-weighted data

Distributions of queue lengths and resource use should reflect not only their values when they increase or decrease, but also the time for which they maintained each of those values. This is done by the use of time-weighted observations. Immediately before an entity is added to or removed from a queue, the simulation records the resource use and/or queue length, x, and, in addition, the time lapse, t, since the last recording was made. The product, xt, gives the time-weighted observation.

5.4.3 Time series data

Time series show the trends in response variables over time. In graphs, the values may be plotted against time. Information must, therefore, be

available about parameter values at regular time intervals. However, discrete-event simulations do not stop at regular time intervals.

The most efficient technique for recording these data depends on the fact that everything in the simulation remains completely stable between discrete events. The program can thus collect data at the beginning or end of each discrete event and make up the data relating to the missing time intervals, in retrospect. The way in which the 'missing' data can be recorded in histograms is described below.

5.5 HISTOGRAMS

A histogram provides a visual representation of a frequency distribution. It may either show the distribution of output variables during the simulation run or else the trends of particular variables over time. Either type of histogram may be displayed while a simulation is running or used to summarize results at the end of a run.

5.5.1 Frequency data

The x-axis is divided into cells of equal width and must provide for the maximum measurement that we are likely to encounter in the simulation (e.g., the largest queue length). When a measurement is made at a particular point in the simulation it will fall in the range of one of the cells on the x-axis.

For observations that are not time-weighted (e.g., waiting time in a queue), a value of one is then added to the frequency value of that cell. This type of histogram is called a *state histogram*.

For time-weighted data, the time since the last recording is added to the cell. Figure 5.2 shows the frequency distribution data of the length of a particular queue using time-weighted data.

5.5.2 Trend data

In Fig. 5.3, the x-axis is divided into cells of equal width, each representing a time interval. The length of the axis must thus provide for the total duration of the simulation run. The y-axis represents the value of the variable (e.g., queue length).

Results are collected from each time beat and the cells of the histogram for each time interval between time beats are made up retrospectively. If n is the number of discrete events since the beginning of the simulation, t_n is the time of the most recent discrete event, and i is the time

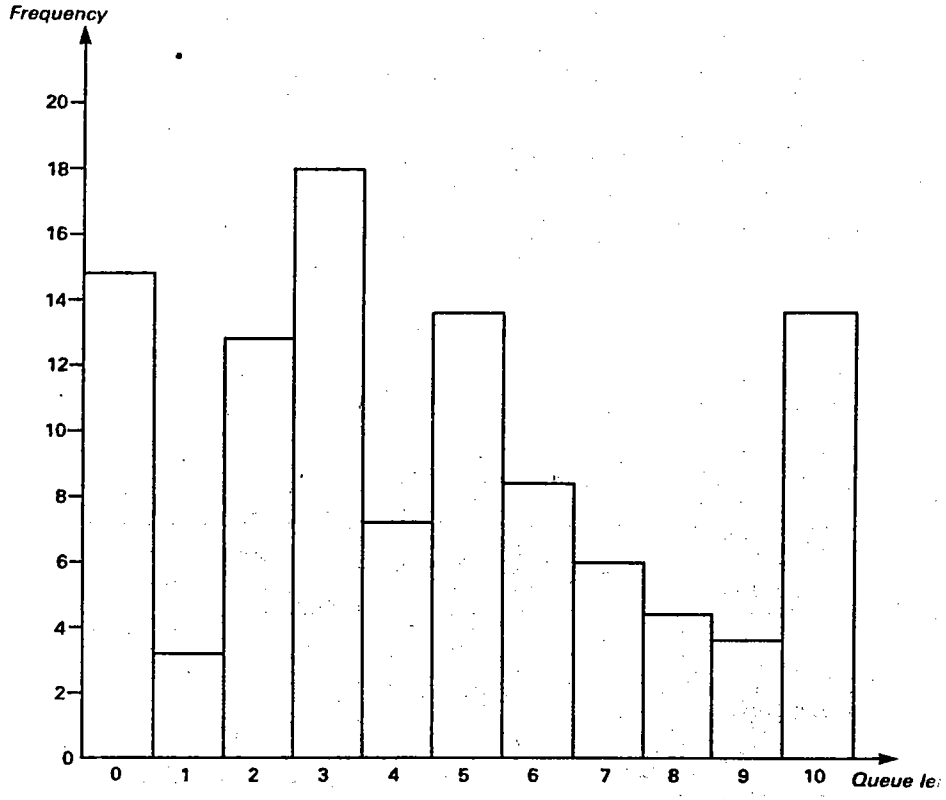

Figure 5.2 Time-weighted histogram showing distribution of queue length in simulation run

interval width, then *trunc* (t_n/i) is the number of time intervals since the beginning of the simulation. If m is the result recorded at time t_{n-1}, then m is recorded in all the cells of the histogram thus

$$trunc\left(\frac{t_{n-1}}{i}+1\right), trunc\left(\frac{t_{n-1}}{i}+2\right), \ldots, trunc\left(\frac{t_n}{i}\right)$$

Figure 5.3 shows a histogram representing the trend in the queue length over time.

5.5.3 Histograms in Pascal_SIM

Pascal_SIM provides some facilities for creating, logging observations to, and printing histograms. The histogram type is a record variable thus:

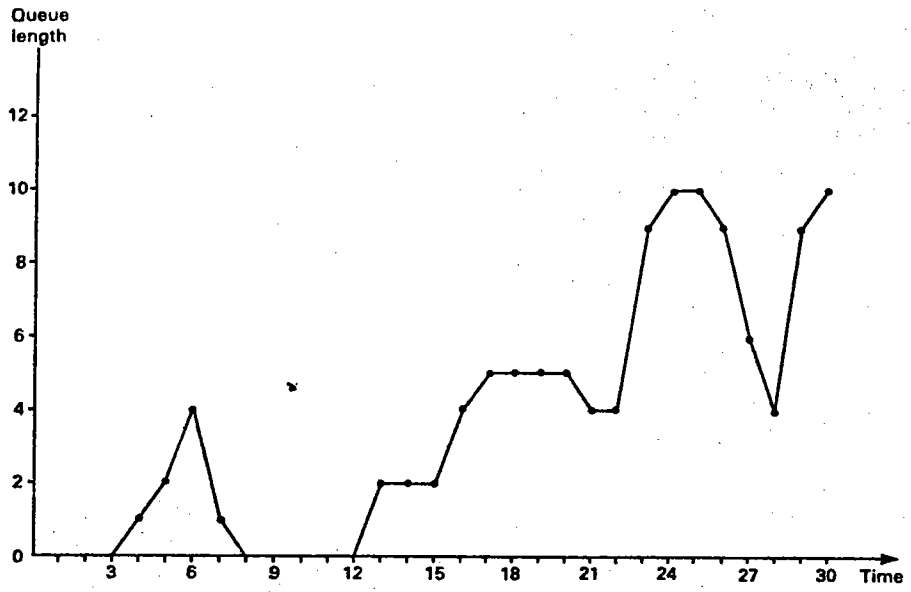

Figure 5.3 Graph showing trend in queue length over simulation run

```
type
  histogram = record
                cell                    :array [cell_num] of real;
                count, width, base,
                  total, sosq,
                  min, max, expended    :real;
                state                   :boolean;
              end;
```

where, using the notation that f_i is the ith frequency value to be recorded and x_i is its value on the x-axis, the various fields are:

(*a*) **cell** is used to store the f_i values in the histogram cells, where the cells are numbered from one up to a maximum number of **max_cell_num**,
(*b*) **count** is the total of the frequencies (f_i) in the histogram (Σf_i),
(*c*) **width** is the width of each cell in the histogram,
(*d*) **base** is the minimum value on the x-axis, i.e., the base value,
(*e*) **total** is the total of each frequency value (f_i) multiplied by x_i, ($\Sigma x_i f_i$),
(*f*) **sosq** is the total of f_i multiplied by x_i-squared, ($\Sigma f_i x_i^2$),
(*g*) **min** is the minimum recorded x_i value,
(*h*) **max** is the maximum recorded x_i value,

(*i*) **expended** gives the last time at which a recording was made, (for use in time-weighted histograms),
(*j*) **state** is switched to *true* for a state or time series histogram and to *false* for a time-weighted histogram.

Histograms are set up in the initialization phase of the simulation using

```
procedure make_histogram (var h :histogram; cell_base,
                          cell_width :real; state :boolean);
```

where *h* was previously declared as a histogram. The variables *cell_base* and *cell_width* set the **base** and **width** fields of the histogram respectively. **Max_cell_num** histogram cells are then established, with the first and last cells for underflow (i.e., less than **base**) and overflow values (i.e., greater than **base** + (**max_cell_num** − 1) * **width**) respectively. The remainder of the histogram fields are initialized to zero using:

```
procedure reset_histogram (var h :histogram);
```

which is also available for use in the simulation program. Data are logged in the histogram using

```
procedure log_histogram (var h :histogram; x,y :real).
```

where x is the distance along the x-axis and y indicates the value to add to the relevant cell in the histogram. Then:

(*a*) for a state histogram, y is added to the relevant cell;
(*b*) for a time-weighted histogram, the relevant cell is updated by (**time** − **expended**) [* y; i.e., the time for which y has persisted and **expended** is then set to **time**;
(*c*) for a time series histogram, x is the number of time intervals since the start of the simulation and y is equal to the variable value (e.g., queue length).

For any type of histogram, these values of x and y are used to update the appropriate **cell**, **count**, **total**, **sosq**, **max** and **min**.

Histograms are printed out on their sides with asterisks to represent frequencies using

```
procedure print_histogram (var pr :text; h :histogram;
                           len :max_string_length);
```

The variable *pr* indicates where the output should be written. Output to the default output stream can be obtained by setting *pr* to *output*; *len* is the maximum number of characters in the height (now going across the page) of the histogram, with a maximum value of **max_string_length**, presently set at 80.

The procedure also calculates and prints out the mean, variance, standard deviation and minimum and maximum x_i values, using the

```
Queue length shown as a time weighted histogram.

mean=   4.04  variance=   9.84  sd =  3.13
min=    0.00  max=  10.00

under   ******************************************          14.53
  1.00  **************                                     5.05
  2.00  ************************************               12.71
  3.00  ***************************************************** 18.12
  4.00  *********************                              7.48
  5.00  ***********************************                13.75
  6.00  ************************                           8.26
  7.00  *****************                                  6.03
  8.00  ***********                                        4.15
  9.00  ********                                           3.52
 10.00  **************                                     12.88
over
```

Figure 5.4 Histogram in Fig. 5.2 represented as output from Pascal_SIM

histogram cell values. For a state histogram, the sum of the cell frequencies (i.e., **count**) is also printed. The histogram is fitted to the width of the page by finding the maximum cell value and scaling it to *len*. Each cell of the histogram is printed out with a number of asterisks in proportion to the frequency value in that cell. Figure 5.4 shows how the histogram shown in Fig. 5.2 would appear, with *len* set to 50; Figure 5.5 shows a Pascal_SIM histogram representing the time series in Fig. 5.3.

5.6 COLLECTION OF RESULTS FOR FURTHER ANALYSIS

Results may be needed for further analysis for statistical tests, summaries of several simulation runs, graphical displays or to provide costings of different simulation outcomes. The advantage of recording detailed statistics as the simulation progresses is that decisions about data analysis can be left until later. The disadvantage is that the recording activity will slow down the progress of the simulation and, without proper selection, can provide a large mass of rather indigestible statistics. The main task of selecting and deciding how to present the information can now take place after the simulation run rather than before.

5.6.1 Recording Information

The measurements that are likely to be of interest are the same as those in the previous section: queue lengths, resource use, and counts of entities in different classes in the system at different points in time. They are collected


```
Queue length over time
processed = 113
mean =   3.40  variance =   10.90  sd =  3.30
min =    4.00 max = 30.00

under
  1.00                                                    0.00
  2.00                                                    0.00
  3.00                                                    0.00
  4.00  *****                                             1.00
  5.00  **********                                        2.00
  6.00  ********************                              4.00
  7.00  *****                                             1.00
  8.00                                                    0.00
  9.00                                                    0.00
 10.00                                                    0.00
 11.00                                                    0.00
 12.00                                                    0.00
 13.00  **********                                        2.00
 14.00  **********                                        2.00
 15.00  **********                                        2.00
 16.00  ********************                              4.00
 17.00  *************************                         5.00
 18.00  *************************                         5.00
 19.00  *************************                         5.00
 20.00  *************************                         5.00
 21.00  ********************                              4.00
 22.00  ********************                              4.00
 23.00  ************************************************  9.00
 24.00  ************************************************** 10.00
 25.00  ************************************************** 10.00
 26.00  ************************************************  9.00
 27.00  ********************************                  6.00
 28.00  ********************                              4.00
 29.00  ************************************************  9.00
 30.00  ************************************************** 10.00
over
```

Figure 5.5 Time series in Fig. 5.3 represented as output from Pascal_SIM

and stored either at each time beat and, where appropriate, weighted by the time lapse between beats. In order to save computer memory where it is insufficiently large, the values may have to be written to disk on each occasion.

5.6.2 Using Information

Information may be presented in tables or histograms using simple formatting instructions and the procedures described in the last section. For

anything more sophisticated, it is sensible to send the information into a specialized statistical, graphics or spreadsheet package. A well-documented package will provide comprehensive information about the format in which it is prepared to accept data.

5.7 ANALYSIS OF RESULTS

The techniques for the analysis of simulation results are based on statistical theory and methods. Some readers may wish to use this section in conjunction with a textbook such as Freud and Walpole (1980).

5.7.1 Independent samples

Results from a stochastic simulation run are always samples from distributions. The results are referred to as *responses* and may be the averages of values collected over the the whole or part of the run, or may be single measurements such as the length of a particular queue at the end of a run. Because responses are samples from distributions, they may vary considerably between runs or between different parts of the same run. The average of the distribution of responses is called the *distribution mean* and is denoted by μ. The extent to which responses vary may be measured by the standard deviation, σ, and the variance, σ^2.

Where a response is the average of values collected from a steady-state simulation (such as an average queue length or average resource use), *one single* response can be used to estimate the *distribution mean*. The longer the period over which the results are averaged, the better the estimate is likely to be. However, *several* independent responses will be needed in order to estimate the *distribution variance*.

Terminating and other non-steady-state simulations invariably need several samples of responses to find the distribution mean as well as the standard deviation and variance.

Most statistical analyses of simulation experiments, therefore, require several independent responses $x_1, \ldots, x_n$. The samples, x_i, are obtained as follows:

(*a*) For a terminating simulation, n different runs of the entire simulation can be used to generate $x_1 \ldots x_n$. For each run, different independent random number streams are used, producing independent estimates of parameters. Sometimes, only part of the duration of the simulation is of interest, so results will only be recorded over that time. Each run is called a *replication*.

(*b*) The method of replications is also used for steady-state simulations. However, each sample is only taken from the steady-state period and not from the run-in period.

(*c*) For a steady-state simulation that takes a long time to reach a steady state, the method of replications can be expensive in computer time. An alternative is the method of *batch means*, where the simulation is executed through the run-in period once, and then values of x_i are obtained from successive time periods of equal length $1 \ldots n$. The method has the disadvantage that considerable correlation may exist between successive samples.

(*d*) Where we are interested in measurements over specific periods or at specific instances, the *regenerative method* is useful. For example, suppose in a manufacturing simulation we are interested in the queue length at a machine when it breaks down. A breakdown can be considered a regenerative point: an independent sample can be obtained immediately after each breakdown.

5.7.2 Means and variances

The parameters likely to be of most interest for summarizing results from simulation runs, are the means and variances. If x_i is the ith response from n replications or batches, then the distribution mean, μ, is estimated by:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

and an unbiased estimate of the variance, σ^2, of the responses becomes:

$$s^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1} \qquad (5.1)$$

5.7.3 Confidence intervals

For predictive simulations, in particular, it is useful to ascertain the accuracy of the estimates of the mean. If the estimate is the sample mean of a set of responses, the *confidence limits* of that estimate give a measure of the accuracy of that estimate; 95% confidence limits are the extreme points of an interval surrounding the sample mean, which have a 95% probability of including the distribution mean.

The variance of the sample mean with sample size, n, is σ^2/n. From (5.1) this is estimated by:

$$s_{\bar{x}}^2 = \frac{\sum_{i=1}^{n} x_i^2}{n(n-1)}$$

The standard deviation is thus estimated by:

$$s_{\bar{x}} = \sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n(n-1)}} \qquad (5.2)$$

If the responses have a Normal distribution, then the sample mean has a Student's t-distribution with $n - 1$ degrees of freedom. However, if the number of runs, n, is large (typically greater than 20), then by the Central Limit Theorem, the sample mean will be Normally distributed, regardless of the distribution of x.

The 95 % confidence limits of the sample mean can be determined from Student's t-tables for small sample numbers. For larger samples which use the Normal distribution, the 95 % confidence limits are:

$$\bar{x} \pm 1{\cdot}96\, s_{\bar{x}}$$

$$\bar{x} \pm 1{\cdot}96 \sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n(n-1)}}$$

5.7.4 Predictive techniques

For non-terminating simulations which do not reach a steady state, it is helpful to measure the mean, $\bar{x}_t$, within a time interval, t, and plot the sampled mean $\bar{x}_t$ against t to give a picture of how the results are varying with time. Several responses can be averaged from different runs to give a more accurate picture. Multiple regression, and other techniques, may be used to fit curves to these values. However, the pattern of $\bar{x}_t$ is often quite complex and may, for example, be cyclic, which can make analysis difficult.

5.7.5 Testing difference between two means

Comparative simulations are designed to test the effect of changing decision variables. Statistical hypothesis testing techniques are useful for determining whether responses from simulations with different decision variable values are statistically significant.

Suppose x is the response variable, with mean μ_x, from the first set of simulation runs and y is the response variable, with mean μ_y, from the comparison set where the decision variable has been changed, then we are testing the hypothesis that $\mu_x = \mu_y$. In statistical notation:

$$H_0 : \mu_x - \mu_y = 0$$
$$H_1 : \mu_x - \mu_y \neq 0$$

If we do n runs with the decision variable set to one value and repeat the runs for the changed decision variable, then the sample mean of the first set of responses is $\bar{x}$ and of the second set of responses is $\bar{y}$. The test is based on the difference between $\bar{x}$ and $\bar{y}$ and the number of standard deviations it is away from the mean. The estimate of the standard deviation depends on whether the random number streams were matched between pairs of comparative runs. If they were, then x_i and y_i cannot be independent and the matched pairs t-test in which $x_i - y_i$ is treated as a single variable, should be used. If they were not, then x_i and y_i may be assumed to be independent, and an appropriate test is the two sample t-test, (usually known simply as the t-test). For large samples, the t-distribution can be approximated to the Normal distribution. Details of these tests can be found in most standard statistics books such as that by Freud and Walpole (1980).

5.7.6 Factor analysis

Factor analysis is an important statistical technique for use in analysing simulation results. Different decision variables which may be of interest include entity arrival rates, the mean time it takes to complete an activity, or the probability of taking a particular branch. These decision variables are *factors*. Clearly there may be very many factors in a complex simulation.

In order to assess the effects of changes in the factors on the simulation output, the simulation is run with different factor values, called *levels*.

Factor analysis is used to measure the extent to which different factors, individually, and interacting with each other, affect simulation output. The complexity of the analysis increases exponentially with each additional factor that is considered because if, for example, there are n factors in which we are interested and factor i is measured at m_i levels, then the number of different possible combinations is

$$\prod_{i=1}^{n} m_i$$

This complexity is further increased if more than one output measure is used.

Factor analysis was originally developed for measuring the effect of varying factors in real observable situations. For example, in an experiment to measure the rate of growth of grass, factors which might be varied in a systematic way are the proportion of rye grass in the seed and the quantity of fertilizer given. The random effects of weather, number of worms and other influences would be unpredictable and more or less uncontrollable. However, in simulation we have considerable control over the random effects and use that control to reduce the variation of the output.

There are therefore several important considerations in using this technique in the analysis of simulation results which distinguish it from its use in real situations. First, each different output measurement will require a different analysis. However, output measurements are likely to be dependent on each other and interpretation is thus confused. If one measurement such as cost can summarize the results from the simulation then only one analysis will be needed and the task is simplified.

Second, the differences between results from comparative runs in a simulation is tightly controlled by giving different distributions in the simulation different random number streams and by the use of variance reduction techniques (see Chapter 8). However, the theory behind statistical tests used in factor analysis assumes either that the results produced at the different factor levels are independent of one another, or that they are matched on all factors apart from the ones of interest, which vary independently of the other . . . either of these is entirely true for most simulation programs.

In conclusion, therefore, factor analysis is suitable for simulations with many factors, each to be tested at several levels. It is time consuming and likely to be inconclusive because statistical tests, normally applied to the results, can no longer be used. However, it is valuable for giving a picture of the effects of factor changes. Readers interested in pursuing this further are recommended to refer to the book by Law and Kelton (1982).

5.8 SUMMARY

A simulation can be *terminating* or *steady state*, or in some instances neither. The start of a steady state can be detected using the *cumulative moving average method*.

Parameter values that are of particular interest in a simulation are queue lengths, waiting times in queues, numbers of entities in the system, the time spent by entities in the system, and resource utilization. These data are collected for presentation in a histogram or for statistical analysis.

Pascal_SIM provides facilities for presenting state and time-weighted histograms, either to show the distribution of particular parameters, or to show their trends with time. Results which are to be analysed statistically should be collected from several simulation runs, or batches within runs, and transferred to a statistical package for further analysis. Techniques include the use of *confidence limits*, tests for the *difference between means* and *factor analysis*.

5.9 EXERCISES

If you are building up the parts of Pascal_SIM as you need them, you will now need to include the histogram routines. When you have implemented these, you will have implemented all the routines in Pascal_SIM necessary to develop non-visual simulations (see Appendix D).

1. List three systems that could be modelled as terminating simulations and three that could be described as non-terminating simulations. In each of the non-terminating simulations explain whether you would expect to achieve a steady state and if so, estimate approximately how long it would take.
2. In the dental clinic example (Exercise 3, Chapter 2):
   (*a*) identify the objectives and the relevant output measures of the simulation model;
   (*b*) describe what histograms should be used and classify them as: state, time weighted, or time series;
   (*c*) if there were no limits on computer time, or your time, describe the analyses you would perform with your simulation program.
3. In the simple hospital system, using the code developed in Exercise 7, Chapter 4, carry out the following tasks.
   (*a*) Find the steady state using the cumulative averages method. Try using different random number streams and vary the average length of stay from 48 to 120 hours to see how the length of the run-in period changes.
   (*b*) Provide histograms of results collected during the steady state showing: queue length, waiting times for admission, and the trend of queue length with time.
   (*c*) Design a series of replications to estimate the average queue length and the confidence limits of that average.
4. Some simulation packages are designed to collect *all* statistics that are likely to be of interest for any simulation program written with that package. The data are written to a file so that they can be analysed at a later date. If you were asked to design such a package, explain what statistics you would collect and when and where in the program (i.e., in the events or the executive) you

would pick them up. Explain how you would organize the file or files of data, bearing in mind that the same program logic would be used for any size or type of simulation model.

5. *Bank System* (Appendix 1.A.1). Plan the output analysis in the following way:
   (*a*) identify the appropriate output measures,
   (*b*) determine what histograms should be provided,
   (*c*) identify the decision variables and suggest what analyses should be performed on several replications of the simulation program.

6. *Traffic Light System* (Appendix 1.A.2). Plan the output analysis for this system in the same way as for the bank system in Exercise 5 above.

6 Hospital and Repair Shop Case Studies

Previous chapters have provided enough background knowledge for developing a simulation program. This chapter presents working simulation programs for both the hospital and repair shop systems and also gives some example output.

6.1 SPECIFYING A SIMULATION STUDY

Chapter 1 showed that producing a relevant and working simulation model is not simply a matter of writing the code. The structure of the simulation model and of the distributions used are derived from the following.

(*a*) *Objectives*. Clear objectives should be laid down prior to the start of the modelling process. Such objectives may be loose and subjective (e.g., investigate the possibility for improved system performance), or may be very detailed (e.g., investigate the effect of increasing the number of machines from 2 to 3). The objectives of the study determine:
   (i) the necessary decision variables,
   (ii) whether visual output (see Chapter 9), a detailed statistics collection, or a mixture of the two is more appropriate, and
   (iii) what output measures are important.

(*b*) *Assumptions*. To model any system, we need to make assumptions about the system behavior. Some assumptions are often accepted implicitly in simulation modelling, for instance, that the system under study is non-adaptive (e.g., doctors do not work at a faster rate when a system becomes busier). While frequently this is not strictly true, such implicit assumptions greatly reduce the complexity of the modelling task.

Also, we often make explicit assumptions for a particular application. Examples of this include assumptions that a transient system always starts and finishes at exactly the same predetermined time, or always starts from the same state (e.g., in a simulation of a bank, it may be assumed that the bank always opens at 9.30 a.m. with no customers waiting for it to open).

Both types of assumption should be documented. The programs should have a flexible design to permit their relaxation at a later stage of development.

(*c*) *Responses.* Chapter 5 discussed the types of parameters and measurements that are most likely to be of interest. If, for instance, we want to measure the trade-off between the provision of a particular resource (e.g., beds or machines), and the time that transient entities (e.g., customers, patients) spend in the system, then summarizing statistics for resource utilization and entity throughput time must be collected.

(*d*) *Decision variables.* The objectives will indicate which factors should remain fixed during a simulation run and which factor levels should be varied. For example, in the hospital system, we may want to vary the number of beds whilst keeping the arrival rate fixed. These variable factors, called decision variables (see Chapter 1), should be made easy to vary. They should be read in to the program at run time, or possibly declared as constants, but certainly not buried in the program as fixed values.

Moreover, a simulation program obviously needs appropriate input data, particularly activity and arrival distributions. We will assume here that data collection and distribution fitting (or distribution assessment in the face of limited or no data) have been carried out. However, it should be noted that this task can be one of the most difficult and most expensive aspects of simulation modelling.

This information, together with the activity diagram, constitutes a specification, from which the simulation pseudo-code and program can be written. Typically, developing the specification can be as time consuming as developing the program, and the data collection can take longer than both of these.

6.2 PROGRAM FOR HOSPITAL SIMULATION

6.2.1 Specification

The hospital system here is a simplification of the activities of a ward of beds, an operating theatre, and the system response to patient arrivals.

Patients arrive for a short hospital stay (hospital-stay-only patients), or for an operation. When beds are not available, those requiring only a stay are given preference to the patients requiring an operation; they stay and then leave. Patients requiring an operation must have a pre-operative stay, then an operation followed by a post-operative stay. They may have to wait for a bed, and then for operating theatre time following their pre-operative stay.

Table 6.1 is an expanded version of Table 1.1. This, together with the activity flow diagram in Chapter 2, is the full specification. There are two major assumptions that need explaining.

(*a*) We are describing patient arrivals as a Poisson process with a constant average rate which is an approximation, because in reality most hospital patients are scheduled to arrive during the

Table 6.1 List providing specification for hospital simulation; used with activity-flow diagram in Chapter 2

Objective
To investigate the effect of bed and operating theatre provision on patient waiting times.

Assumptions
The system runs continuously, with no breaks for weekends, holidays, etc.
Arrivals are subject to a Poisson process evenly distributed throughout the day.
Hospital-stay patients are admitted in preference to patients requiring operations.

Decision variables
Number of beds.
Opening times of operating theatre.

Responses
Number of each type of patient waiting to be admitted.
Utilization of beds.
Waiting-time for operation, following completion of pre-operative stay.

Activity durations
Time between hospital stay arrivals:
negative exponential, mean = 12 hours.
Time between operation arrivals:
negative exponential, mean = 6 hours.
Time for hospital stay:
negative exponential, mean = 2·5 days.
Time for pre-operative stay:
negative exponential, mean = 1 day.
Time for post-operative stay:
negative exponential, mean = 3 days.
Time for operation:
normal, mean = 0·75 hour, standard deviation = 0·25 hour.

day. However, it provides a first step in the process of investigating the effects of changing bed provision and operating theatre time.

(*b*) The system is being considered as operating continuously, whereas, in real life for example, theatre time may not be scheduled for weekends and arrivals may not occur at night.

We are, thus, considering the system to be steady-state, and will investigate the effect of changing decision variables on steady-state parameters. In a full study, the effects of these assumptions would be assessed as part of the validation process (see Chapter 8).

The most important measurements from the simulation (and this is true in many health care simulations) are bed utilization and the lengths of waiting lists. They give useful information about the extent to which the number of beds and the availability of the operating theatre constrain arrivals into the system. Another important measure is the length of time that patients requiring an operation have to wait for one, following their pre-operative stay. This will indicate where there is a shortfall of operating theatre time.

6.2.2 Program details

Together with the pseudo-code for the hospital, developed in Chapter 2, and the specification (comprising the activity flow diagram and Table 1.1), we can start to develop the program. This chapter provides the three-phase and event-scheduling versions of the program. The process view version is considered in Chapter 12.

The theatre can be scheduled for two types of event: the end of a patient's operation, and the time at which it closes. The pseudo-code in Chapter 2 indicates how this can be programmed. The entity representing the theatre is always in the calendar to determine when the theatre should be opened or closed. A Boolean variable showing whether the theatre is available or not is changed when an operation starts or finishes. (For a more elegant and general approach to handling this problem, see Chapter 11).

Histograms must be declared, and new values logged each time there is a change in a bound or conditional event. The waiting time for an operation has to be calculated before it can be recorded. However, in *start_operation*, we know the time that the patient finished the pre-operative stay because the patient entity record will still be marked with the calendar time. The waiting time is thus the present clock time tim minus current_time.

6.2.3 Program

The program is shown in Appendix 6.A. The decision variables are all declared as global constants: the number of beds, the length of time the theatre is open, and the length of time it is shut, which are called *amount_of_beds*, *theatre_open_time*, and *theatre_closed_time*, respectively. They are presently set to 20, 4, and 24. The time base of the simulation is hours. The theatre is declared as a record thus:

```
var
  theatre :record
             body              :entity;
             open, available   :boolean;
           end;
```

where *open* will be true when the theatre is open, and *available* will be true if there is no operation in progress. An operation can continue past the scheduled theatre closing time, but no further operations will take place after this until the theatre is reopened.

Notice that q4 is a dummy queue, since there are no resource constraints on the post-operative stay activity. Thus the conditional event *start_post_operative_stay* is superfluous: the patient could be scheduled for a post-operative stay within the *end_operation* bound event. However, keeping this separate, as was done with the pseudo-code in Chapter 2, has advantages. Every activity is defined by a pair of conditional and bound events, making the program easier to follow and modify.

Histograms are declared to gather data from: the *hospital-stay-only* queue, the *operation-only* queue, bed utilization, and the waiting time for an operation following pre-operative stay. These are: *q1_hist*, *q2_hist*, *bed_util*, and *op_waiting*, respectively.

6.2.4 Run-in period

As the hospital simulation is to be treated as a steady-state simulation, it is necessary to detect the start of the steady state (the end of the initial transient period) to determine when to start collecting statistics.

The detection of the start of the steady state, using the cumulative moving-average method as discussed in Chapter 5, is quite easy to program. Table 6.2 shows some intermediate cumulative mean values for the output parameters. These were obtained by adding code to the B phase part of the executive to produce the means of the response variables every 48 simulated hours, after the first 96 hours had elapsed.

Figure 6.1 shows Table 6.2 plotted using a spreadsheet package. It is interesting to note in Fig. 6.1, that the *hospital-stay only* queue, and the

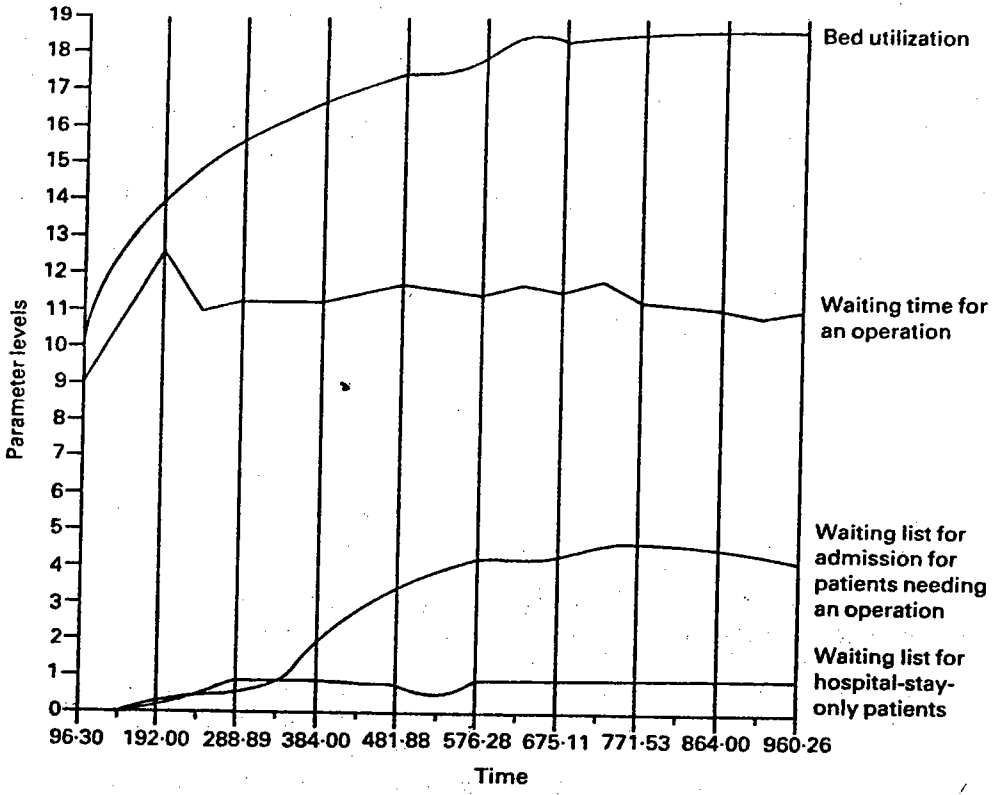

Figure 6.1 Steady-state detection for hospital simulation using cumulative means in Table 6.2; graph was produced using Lotus 1–2–3

waiting time for an operation, reach steady states quite quickly, unlike the *operation-only* queue and bed utilization, which take considerably longer. This is to be expected, since *operation-only* patients are only admitted if no *hospital-stay-only* patients are waiting.

Figure 6.1 suggests that the run-in period should be about 720 hours, hence this number is used in the program in Appendix 6.A. In a real project, a number of cumulative-average measures would be produced for a number of output measures using different random number streams, to ensure that a steady state had been reached.

Rather than starting the collection of statistics after the run-in period, it is started at the beginning of the simulation, and then all histograms are reset after the run-in. Although this approach might seem inefficient, it is easier to program and is unlikely to make the program take much more time to run.

Table 6.2 Cumulative means for hospital output parameters. First mean is for first event after 96 hours (i.e., four days) of simulated time; means are then every 48 hours

Time	*Hospital-stay queue length*	*Operation-only queue length*	*Bed utilization*	*Operation waiting time*
96·30	0·00	0·00	10·14	8·92
144·07	0·00	0·00	11·92	11·06
192·00	0·16	0·08	13·84	12·84
241·33	0·21	0·33	14·99	10·50
288·89	0·99	0·50	15·94	10·94
336·29	1·09	1·45	16·50	10·91
384·00	1·09	2·12	16·90	10·91
433·84	1·12	2·75	17·29	11·12
481·88	1·12	3·44	17·57	11·91
528·00	1·04	4·02	17·79	11·53
576·28	0·99	4·52	17·97	11·30
628·73	1·04	4·98	18·31	11·88
675·11	1·04	5·20	18·25	11·39
720·00	1·02	5·62	18·38	11·86
771·53	1·03	5·76	18·47	11·06
816·09	1·05	5·81	18·57	10·69
864·00	1·05	5·85	18·65	10·76
914·14	1·06	5·73	18·69	10·44
960·26	1·08	5·34	18·66	10·79

6.2.5 Results

Following the run-in period, the program runs for 14 simulated days: Figure 6.2 shows the output statistics from a run. These are based on a different set of random numbers from those used to determine the run-in period.

The distribution of queue length for *hospital-stay-only* patients, with a mean of only 1·17, has a large variance, as does the queue for patients requiring an operation, which has a mean of 4·58. Bed utilization is 19·93, all 20 beds being in use for 318 simulated hours of the 336 hours of replication; 26 patients received an operation, and waiting time was again highly variable.

As discussed in Chapter 5, each run with a different set of random number streams produces a single replication. Further replications are necessary to provide an accurate estimate of output parameters. The excessive variance in the output measures needs to be dealt with using *variance reduction* techniques, which will be covered in Chapter 8.

The mean values in the replication shown in Fig. 6.2 differ from the

```
Number of Hospital Stay Patients Waiting for Admission

mean =    1.17  variance =    1.34  sd =    1.16
min =     0.00  max =    5.00

under   ************************************************************ 120.81
  1.00  ***********************************************              92.39
  2.00  ***********************************                         67.91
  3.00  *****************                                            33.52
  4.00  ******                                                       12.57
  5.00  |                                                             0.77
  6.00                                                               0.00
  7.00                                                               0.00
  8.00                                                               0.00
  9.00                                                               0.00
 10.00                                                               0.00
 11.00                                                               0.00
 12.00                                                               0.00
 13.00                                                               0.00
 14.00                                                               0.00
 15.00                                                               0.00
over                                                                 0.00

Number of Operation Patients Waiting for Admission

mean =    4.58  variance =    6.81  sd =    2.61
min =     0.00  max =    9.00

under   ***************************                                 41.17
  1.00  ***************                                              23.57
  2.00  ***                                                           6.27
  3.00  *****************                                            26.81
  4.00  ************************************                         56.18
  5.00  ************************                                     36.91
  6.00  *************                                                19.52
  7.00  ************************************************************ 96.38
  8.00  ********                                                     13.70
  9.00  *****                                                         8.93
 10.00                                                               0.00
 11.00                                                               0.00
 12.00                                                               0.00
 13.00                                                               0.00
 14.00                                                               0.00
 15.00                                                               0.00
over                                                                 0.00
```

cumulative means shown in Table 6.2: two values are above the final cumulative mean; two are below. Due to the inherent variance in the simulation,


```
Utilization of Beds

mean =   19.93  variance =    0.14  sd =    0.37
min =    17.00  max =   20.00

under                                                                0.00
  1.00                                                               0.00
  3.00                                                               0.00
  5.00                                                               0.00
  7.00                                                               0.00
  9.00                                                               0.00
 11.00                                                               0.00
 13.00                                                               0.00
 15.00                                                               0.00
 17.00  *                                                            7.11
 19.00  ************************************************************ 318.00
 21.00                                                               0.00
 23.00                                                               0.00
 25.00                                                               0.00
 27.00                                                               0.00
 29.00                                                               0.00
over                                                                 0.00

Waiting time for Operation
processed =    26
mean =    9.40  variance =   56.49  sd =    7.52
min =     0.00  max =   22.97

under   ************************************************************  6.00
  1.00  ********************************************                 4.00
  4.00                                                               0.00
  7.00  ********************************                             3.00
 10.00  ********************************                             3.00
 13.00  *********************************************                4.00
 16.00  ********************************                             3.00
 19.00  **********************                                       2.00
 22.00  **********                                                   1.00
 25.00                                                               0.00
 28.00                                                               0.00
 31.00                                                               0.00
 34.00                                                               0.00
 37.00                                                               0.00
 40.00                                                               0.00
 43.00                                                               0.00
over
```

Figure 6.2 Output from run of hospital simulation with 20 beds and with operating theatre open for four hours a day

results from different replications may differ widely from each other, and from the cumulative means. However, we should expect accurate estimates

of steady-state parameters, using estimates based on several runs, to be larger than the final cumulative mean values, since these include statistics from the initial transient period.

6.2.6 Event-scheduling version

The pseudo-code for an event-scheduling version of the hospital system was presented in Chapter 2. The equivalent program is shown in Appendix 6.B. Statistics collection facilities, which would be much the same as in the three-phase version shown in Appendix 6.A, have been omitted. Note that the executive differs from that for the three-phase approach only in that the scan of the conditional events has been removed.

6.3 PROGRAM FOR REPAIR SHOP SIMULATION

6.3.1 Specification

The repair shop consists of a number of machines, each of which can breakdown and need repair. This is done by a mechanic, who first removes any material stuck in the machine and all covers, obtains the equipment necessary to repair the machine, and finally repairs it. After this, the machine can return to work. Machines are repaired in the order in which they breakdown and if no set of equipment is available, a mechanic will wait until one becomes available rather than start to remove the covers on any other broken machine.

Each production shift works for eight hours, and there is sufficient time between shifts for the mechanics to finish repairing any machines that are still down following the conclusion of the shift. Thus mechanics, if necessary, work overtime so as to complete the repairs.

Table 6.3, which is an extension of Table 1.2, indicates that the simulation is to be used as an experimental tool to investigate the relationship between downtime and resource provision. The main output measure is therefore downtime, and the decision variables are the levels of resources, mechanics and equipment. Although the *acceptable level* of downtime is not prescribed in the simulation objective, typically this would be clarified in discussions with the client.

Important additional measures are the utilization of both mechanics and equipment, since these will indicate where resources should be added or withdrawn so as to achieve an acceptable balance.

Although the simulation has short activity duration times, and may reach a steady state quite quickly, summarizing statistics for the duration

Table 6.3 List providing specification for repair shop simulation; used with activity flow diagram in Chapter 2

Objective
To investigate the effect of mechanic and equipment provision on machine down-time.

Assumptions
The shift starts 'cold' at time 0.
The shift ends after eight hours but the mechanics continue working until all the machines are repaired.
There is always enough material to keep a working machine busy.
The probability of breakdown is equal for all machines at the start of each shift.
The time taken for mechanics to travel to a machine, or transport equipment, is negligible.

Decision Variables
Number of mechanics.
Number of sets of equipment.

Responses
Machine down-time.
Utilization of mechanics.
Utilization of equipment.
Post-shift working.

Activity Durations
Time between machine breakdowns:
negative exponential, mean = 60 minutes.
Time to do removal: normal, mean = 5 minutes, sd = 1 minute.
Time to do repair: normal, mean = 8 minutes, sd = 2 minutes.

of entire shifts are very relevant. The simulation should thus be treated as a terminating simulation. Additionally, the length of time that mechanics must work after the end of the shift is important, since labour costs will be associated with this time. If costs were to be applied to these measures, one cost figure could summarize the results from all these output measures.

This specification includes two important assumptions which require explanation.

(*a*) It is assumed that the probability of breakdown for all machines is equal at the start of the shift, although some machines may have been recently repaired and others not. As the mean time between breakdowns is fairly small – only one hour – we are assuming that the extra accuracy to be gained from carrying through sampled breakdown times from one shift to another is not important.

(*b*) Although the mechanics may spend time moving equipment to

machines or waiting for spare parts, we are choosing to ignore this so as to keep the model simple.

Both these assumptions could be lifted if the resulting simulation model were to prove insufficiently accurate.

If we were to extend the problem by considering the number of machines to be a decision variable, the simulation structure would still be relevant. The logic of the system is independent of the number of machines, so this can be increased or reduced as desired. It is sometimes useful to program such variables as if they were *decisions variables* since, in any simulation study, fixed parameters may have to be varied at a later date. However, if we were to relax the second assumption and take account of the time that mechanics spend moving between machines, it would affect the simulation structure and necessitate the introduction of new activities.

6.3.2 Program details

The program is developed using the pseudo-code presented in Chapter 2. Every machine must be created and entered in the calendar at the start of a run in order to set the breakdown times. Thus the **initialize** procedure should be:

```
procedure initialize;
  begin
    < make the calendar and streams >;
    < make the bins >;
    < make the queues >;
    < make the machines >
    for < all machines > do < cause breakdown time >;
  end;
```

The program must be adjusted to take account of the run length of the simulation. It is not sensible to run the simulation simply for an eight-hour stretch, since we wish to measure the amount of overtime that the mechanics work. We need to run the simulation out, that is, run it until all the machines that are queueing for repair, or are in the process of being repaired, are completely finished after the end of the eight-hour shift. This can be done by running the simulation for eight hours, setting a flag that indicates when the shift has finished, and then running the simulation again until all repairs are complete. The flag is used to stop any activities except the completion of existing repairs. The run will come to an orderly halt when the calendar becomes empty. If we use a Boolean flag called *at_work*, the main program can be written thus:


```
begin
  initialize;
  run (8*60,2);          { run for 8 hour shift }
  at_work := false;      { close down the shift }
  run (10000,2);         { run out the remaining repairs }
  report;
end.
```

where 10000 minutes is an arbitrary long time, longer than any repair is likely to take. So *at_work* can be employed in *end_working* and *end_repair* thus:

```
procedure end_working;
  begin
    if at_work then
      < give machine to tail of q1 >;
  end;

procedure end_repair;
  begin
    < return the mechanic >;
    < return the equipment >;
    if at_work then
      < cause a new breakdown time >;
  end;
```

The statistics measures for the output parameters are time-weighted (see Chapter 5). The histograms showing the utilization rates for the mechanics and equipment are built up by logging the number of unavailable resources to them each time there is a change in resource levels (i.e., before **acquire** or **return**).

Whereas the simulation automatically keeps track of the number of resources available, counts of entities in one state rather than another have to be kept explicitly. To collect statistics on machine downtime, the count is changed and logged to the histogram each time that a machine fails or comes back into use.

6.3.3 Program

Following the specification, the pseudo-code, and the details above, most of the program falls into place. The program is shown in Appendix 6.C. The global variables that must be declared are the *at_work* flag, the queues *q1* and *q2*, the bins *mechanic* and *equipment*, and the three necessary histograms. As with the hospital example, it is good practice to declare the

resource levels and all stream numbers as global constants, so that these can be altered easily during experimentation.

6.3.4 Results

Figure 6.3 shows the results from one run of the repair shop simulation. Again, this is a *single replication*, and more replications will be necessary to give an accurate estimate of output parameters and comparative results.

At the end of the shift, four machines were still broken, the last of which had its repair finished at 12·66 minutes after the shift ended. The summarizing statistics show that the number of machines down varied between 0 and 10, with a mean of 4·54. The utilization of mechanics was higher than for the equipment (84·25 % against 68·7 %), but note that for a large part of the time (334·72 minutes, i.e., over 5·5 hours) all the mechanics were busy.

```
Finished repair at   0: 8: 1.02
Finished repair at   0: 8: 7.31
Finished repair at   0: 8:10.29
Finished repair at   0: 8:12.66

Number of Machines Broken Throughout Shift

mean =    4.54  variance =    4.79  sd =    2.19
min  =    0.00  max  =   10.00

under   ***********                                                  20.57
 1.00   ***************                                              24.56
 2.00   ************************                                     39.65
 3.00   *********************************                            53.02
 4.00   ****************************************************************** 104.21
 5.00   ****************************************************           83.42
 6.00   ************************************                         63.37
 7.00   ************************                                     41.45
 8.00   ****************                                             29.11
 9.00   ***********                                                  19.28
10.00                                                                 0.26
11.00                                                                 0.00
12.00                                                                 0.00
13.00                                                                 0.00
14.00                                                                 0.00
15.00                                                                 0.00
over                                                                  0.00
```



```
Utilization of Mechanics

mean =    3.37  variance =    1.18  sd =    1.09
min  =    0.00  max  =    4.00

under   **                                                           13.54
 1.00   *******                                                      40.51
 2.00   *****                                                        29.25
 3.00   *************                                                74.64
 4.00   ********************************************************** 334.72
 5.00                                                                 0.00
 6.00                                                                 0.00
 7.00                                                                 0.00
 8.00                                                                 0.00
 9.00                                                                 0.00
10.00                                                                 0.00
11.00                                                                 0.00
12.00                                                                 0.00
13.00                                                                 0.00
14.00                                                                 0.00
15.00                                                                 0.00
over                                                                  0.00

Utilization of Equipment

mean =    2.06  variance =    0.84  sd =    0.92
min  =    0.00  max  =    3.00

under   *********                                                    29.99
 1.00   ********************************                            102.47
 2.00   ****************************************************        168.20
 3.00   ********************************************************** 192.00
 4.00                                                                 0.00
 5.00                                                                 0.00
 6.00                                                                 0.00
 7.00                                                                 0.00
 8.00                                                                 0.00
 9.00                                                                 0.00
10.00                                                                 0.00
11.00                                                                 0.00
12.00                                                                 0.00
13.00                                                                 0.00
14.00                                                                 0.00
15.00                                                                 0.00
over                                                                  0.00
```

Figure 6.3 Output from run of repair shop simulation with four mechanics and three sets of equipment

6.4 SUMMARY

In this chapter, we have developed complete programs for both the hospital and repair shop systems, using the three-phase approach. Each program was based upon the pseudo-code developed in Chapter 2, and a specification of the simulation study. The hospital simulation has also been coded using the event-scheduling approach. The hospital was treated as a *steady-state* simulation; the repair shop was treated as a *terminating* simulation. A single replication of each has been shown.

The models presented here now require *validation* prior to their use in *experimentation*. The process of validation, variance reduction and experimentation is developed in Chapter 8. Chapter 10 shows how the two case studies may be provided with visual output while Chapter 12 provides a process-view version of the hospital system.

6.5 EXERCISES

PascalSIM should now be implemented with all the routines required for non-visual simulations (see Appendix D).

1. *Hospital System*. Implement the program described in this chapter, running the steady state analysis and providing the histograms described.
   (*a*) Try running the program with different random number streams and compare the results.
   (*b*) Using six replications, derive confidence limits of the estimate of the average bed occupancy and theatre utilization. (Note: the replications should be independent of each other.)
   (*c*) Using one of the replications, vary the decision variables to see how they affect the output. Summarize your results graphically, either by hand, or by using a spreadsheet, statistics, or graphics package.
2. *Repair Shop System*. Implement the program described in this chapter.
   (*a*) Try running the program with different random number streams and compare the results.
   (*b*) Using 12 replications, derive confidence limits of the estimate of the machine down-time and post-shift working. (Note: the replications should be independent of each other.)
   (*c*) Using one of the replications, vary the decision variables to see how they affect the output. Summarize your results graphically.

   Explain what additional information the management would need in order to arrive at a decision about the number of mechanics to be employed in this workshop.
3. A cell is a self-contained part of a manufacturing operation. Raw materials or part-finished products enter the cell. Part or completely finished products leave the cell.

   A small manufacturing company has a cell composed of three work stations each of which is fed by a buffer that can hold three products. Products arriving at a cell attempt to enter each of the three buffers in turn but if all the buffers are full, they leave without being processed.

   A data collection exercise found that the inter-arrival time distribution of products at a cell had a negative exponential distribution with a mean of 15 minutes. The time taken to process a product was Normally distributed with a mean of 30 minutes and a standard deviation of 5 minutes.
   (*a*) Produce a simulation model of the manufacturing cell.
   (*b*) Estimate the duration of the initial transient period and, hence, the start of the steady state.
   (*c*) Using the method of replications, produce a confidence interval for the utilization of each work station.
4. *Bank Case Study* (Appendix 1.A.1). Carry out the following tasks.
   (*a*) Prepare a specification for the case study.
   (*b*) Using the code developed in the exercises of previous chapters, write programs (for both the four-queue and the one-queue system) to meet this specification.
   (*c*) Perform several replications of each program and compare the results. Discuss these in the light of the assumptions you made in the specification.
5. *Traffic Light System* (Appendix 1.A.2). Carry out the following tasks.
   (*a*) Prepare a specification for the case study.
   (*b*) Using the code developed in the exercises of previous chapters, write a program to meet this specification.
   (*c*) Perform several replications of the program, using the present evening rush-hour data, to estimate the key output measures. Vary the decision variable(s), using one or more replications, and discuss the results.

* * *

6.A HOSPITAL SIMULATION USING THREE-PHASE APPROACH

```
program simulation (output);

const
  amount_of_beds      = 20;
  patient1_seed       = 1;
  patient2_seed       = 2;
  hospital_stay_seed  = 3;
  pre_op_stay_seed    = 4;
  operation_seed      = 5;
```

```
    post_op_stay_seed   = 6;
    theatre_open_time   = 4;
    theatre_closed_time = 20;

  var
    bed                       :bin;
    q1,q2,q3,q4               :queue;
    theatre                   :record
                                 body           :entity;
                                 open,available :boolean;
                               end;
    q1_hist,q2_hist,
      bed_util,op_waiting     :histogram;

  { B events }
  procedure patient1_arrives; { stay } { B1 }
    begin
      log_histogram (q1_hist,count(q1),1);
      give_tail (q1,current);
      cause (1,new_entity(1,1),negexp(6,patient1_seed));
    end;

  procedure patient2_arrives; { operation } { B2 }
    begin
      log_histogram (q2_hist,count(q2),1);
      give_tail (q2,current);
      cause (2,new_entity(2,1),negexp(12,patient2_seed));
    end;

  procedure end_hospital_stay; { B3 }
    begin
      with bed do log_histogram (bed_util,number-num_avail,1);
      return (bed,1);
      dis_entity (current);
    end;

  procedure end_pre_operative_stay; { B4 }
    begin
      give_tail (q3,current);
    end;

  procedure end_operation; { B5 }
    begin
      theatre.available := true;
      give_tail (q4,current);
    end;

  procedure end_post_operative_stay; {B6 }
    begin
      with bed do log_histogram (bed_util,number-num_avail,1);
      return (bed,1);
      dis_entity (current);
    end;
```



```
  procedure open_theatre; { B7 }
    begin
      theatre.open := true;
      cause (8,current,theatre_open_time);
    end;

  procedure close_theatre; { B8 }
    begin
      theatre.open := false;
      cause (7,current,theatre_closed_time);
    end;

  { C events }
  procedure start_hospital_stay; { C1 }
    begin
      while (bed.num_avail>0) and (not empty(q1)) do
        begin
          with bed do
            log_histogram (bed_util,number-num_avail,1);
          acquire (bed,1);
          log_histogram (q1_hist,count(q1),1);
          cause (3,take_top(q1),negexp(60,hospital_stay_seed));
        end;
    end;

  procedure start_pre_operative_stay; { C2 }
    begin
      while (bed.num_avail>0) and (not empty(q2)) do
        begin
          with bed do log_histogram (bed_util,number-num_avail,1);
          acquire (bed,1);
          log_histogram (q2_hist,count(q2),1);
          cause (4,take_top(q2),negexp(24,pre_op_stay_seed));
        end;
    end;

  procedure start_operation; { C3 }
    begin
      with theatre do
        while open and available and (not empty(q3)) do
          begin
            available := false;
            current := take_top (q3);
            log_histogram (op_waiting,tim-current^.time,1);
            cause (5,current,normal(0.75,0.25,operation_seed));
          end;
    end;
```

```
procedure start_post_operative_stay; { C4 }
  begin
     while (not empty(q4)) do
       begin
         cause (6,take_top(q4),negexp(72,post_op_stay_seed));
       end;
  end;

procedure display;
  begin
  end { display };

procedure run(duration:real;max_C:cardinal);
  var
    c :cardinal;
  begin
    running := true;
    repeat
      if calendar=calendar^.next then running := false
        else
          begin
            display;
            tim := calendar^.next^.item^.time;
            if duration<tim then running := false
              else
                begin
                  while (calendar<>calendar^.next) and
                        (tim=calendar^.next^.item^.time)  do
                    begin
                      calendar_top;
                      case current^.next_B of
                        0: ;
                        1: patient1_arrives;
                        2: patient2_arrives;
                        3: end_hospital_stay;
                        4: end_pre_operative_stay;
                        5: end_operation;
                        6: end_post_operative_stay;
                        7: open_theatre;
                        8: close_theatre;
                      end;
                    end;
                  for c := 1 to max_C do
                    case c of
                      1: start_hospital_stay;
                      2: start_pre_operative_stay;
                      3: start_operation;
                      4: start_post_operative_stay;
                    end;
                end;
          end
    until not running;
  end {run}.
```



```
procedure initialize;
  begin
    make_sim;
    make_streams;
    make_bin (bed,amount_of_beds);
    make_queue (q1); make_queue (q2);
    make_queue (q3); make_queue (q4);
    with theatre do   {create theatre}
      begin
        body := new_entity (3,1);
        open := true; available := true;
        cause (8,body,theatre_closed_time);
      end;
    make_histogram (q1_hist,1,1,false);
    make_histogram (q2_hist,1,1,false);
    make_histogram (bed_util,1,2,false);
    make_histogram (op_waiting,1,3,true);
  end { initialize };

procedure report;
  begin
    writeln
   ('Number of Hospital Stay Patients Waiting for Admission');
    print_histogram (output,q1_hist,60);
    writeln;
    writeln
   ('Number of Operation Patients Waiting for Admission');
    print_histogram (output,q2_hist,60);
    writeln;
    writeln ('Utilization of Beds');
    print_histogram (output,bed_util,60);
    writeln;
    writeln ('Waiting time for Operation');
    print_histogram (output,op_waiting,60);
  end { report };

begin
  initialize;
  cause (1,new_entity(1,1),0);
  cause (2,new_entity(2,1),0);
  run (24*30,4);       { 30 days run in period }
  reset_histogram (q1_hist);
  reset_histogram (q2_hist);
  reset_histogram (bed_util);
  reset_histogram (op_waiting);
  run (24*(30+14),4);    { do a replication of 14 days }
  report;
end.
```

6.B HOSPITAL SIMULATION USING EVENT METHOD

```
program simulate (output);

const
  amount_of_beds       = 20;
  patient1_seed        = 1;
  patient2_seed        = 2;
  hospital_stay_seed   = 3;
  pre_op_stay_seed     = 4;
  operation_seed       = 5;
  post_op_stay_seed    = 6;
  theatre_open_time    = 4;
  theatre_closed_time  = 20;

var
  bed       :bin;
  q1,q2,q3  :queue;
  theatre   :record
              body              :entity;
              open,available    :boolean;
            end;

{ 8 events }
procedure patient1_arrives;  { stay } { B1 }
  begin
    cause (1,new_entity(1,1),negexp(6,patient1_seed));
    if (bed.num_avail>0) then
      begin
        acquire (bed,1);
        cause (3,current,negexp(60,hospital_stay_seed));
      end
    else give_tail (q1,current);
  end;

procedure patient2_arrives;  { operation } { B2 }
  begin
    cause (2,new_entity(2,1),negexp(12,patient2_seed));
    if (bed.num_avail>0) then
      begin
       acquire (bed,1);
       cause (4,current,negexp(24,pre_op_stay_seed));
      end
    else give_tail (q2,current);
  end;

procedure end_hospital_stay;  { B3 }
  begin
    return (bed,1);
    dis_entity (current);
    if not empty (q1) then
      begin
```



```
        acquire (bed,1);
        cause (3,take_top(q1),negexp(60,hospital_stay_seed))
      end
    else
      if not empty (q2) then
        begin
          acquire (bed,1);
          cause (4,take_top(q2),negexp(24,pre_op_stay_seed));
        end;
  end;

procedure end_pre_operative_stay;  { B4 }
  begin
    with theatre do
      if open and available then
        begin
          available := false;
          cause (5,current,normal(0.75,0.25,operation_seed));
        end
      else give_tail (q3,current);
  end;

procedure end_operation;  { B5 }
  begin
    with theatre do
      begin
        available := true;
        cause (6,current,negexp(72,post_op_stay_seed));
        if open and (not empty (q3)) then
          begin
            available := false;
            cause (5,take_top(q3),
                   normal(0.75,0.25,operation_seed));
          end;
      end;
  end;

procedure end_post_operative_stay;  { B6 }
  begin
    return (bed,1);
    dis_entity (current);
    if not empty (q1) then
      begin
        acquire (bed,1);
        cause (3,take_top(q1),negexp(60,hospital_stay_seed));
      end
    else
      if not empty (q2) then
        begin
          acquire (bed,1);
          cause (4,take_top(q2),negexp(24,pre_op_stay_seed));
        end;
  end;
```

```
procedure open_theatre;  { B7 }
  begin
    with theatre do
      begin
        open := true;
        cause (8,current,theatre_open_time);
        if not empty (q3) then
          begin
            available := false;
            cause (5,take_top(q3),
            normal(0.75,0.25,operation_seed));
          end;
      end;
  end;

procedure close_theatre;  { B8 }
  begin
    theatre.open := false;
    cause (7,current,theatre_closed_time);
  end;

procedure run (duration :real);
  begin
    running := true;
    repeat
      if calendar=calendar^.next then running := false
        else
          begin
            tim := calendar^.next^.item^.time;
            if duration<tim then running := false
              else
                while (calendar<>calendar^.next) and
                      (tim=calendar^.next^.item^.time)  do
                  begin
                    calendar_top;
                    case current^.next_B of
                      0: ;
                      1: patient1_arrives;
                      2: patient2_arrives;
                      3: end_hospital_stay;
                      4: end_pre_operative_stay;
                      5: end_operation;
                      6: end_post_operative_stay;
                      7: open_theatre;
                      8: close_theatre;
                    end;
                  end;
          end
    until not running;
  end { run };
```



```
procedure initialize;
  begin
    make_sim;
    make_streams;
    make_bin (bed,amount_of_beds);
    make_queue (q1);
    make_queue (q2);
    make_queue (q3);
    with theatre do    { create theatre }
      begin
        body := new_entity (3,1);
        open := true; available := true;
        cause (8,body,theatre_closed_time);
      end;
  end { initialize };

procedure report;
  begin
  end { report };

begin
  initialize;
  cause (1,new_entity(1,1),0);
  cause (2,new_entity(2,1),0);
  run (24*30);
  report;
end.
```

6.C REPAIR SHOP SIMULATION

```
program simulate (output);

const
  amount_of_machines    = 20;
  amount_of_mechanics   = 4;
  amount_of_equipments  = 3;
  inter_break_down_time = 60;
  breakdown_seed        = 11;
  removal_time_seed     = 12;
  repair_time_seed      = 13;

var
  at_work                          :boolean;
  q1,q2                            :queue;
  mechanic, equipment              :bin;
  mechanic_util, equipment_util    :histogram;
  broken                           :record
                                      number  :cardinal;
                                      content :histogram;
                                    end;
```

```
{ Bound events }
procedure end_working;  { breakdown } { B1 }
  begin
    if at_work then { machine can't break down after shift }
      begin
        with broken do
          begin
            number := number+1;
            log_histogram (content,number,1);
          end;
        give_tail (q1,current);
      end;
  end;

procedure end_removal;  { B2 }
  begin
    give_tail (q2,current);
  end;

procedure end_repair;  { B3 }
  begin
    with equipment do
      log_histogram (equipment_util,number-num_avail,1);
    with mechanic do
      log_histogram (mechanic_util,number-num_avail,1);
    return (mechanic,1);
    return (equipment,1);
    if at_work then
      begin
        cause (1,current,negexp(inter_break_down_time,
        breakdown_seed));
        with broken do
          begin
            number := number-1;
            log_histogram (content,number,1);
          end;
      end
      else
        begin
          write ('Finished repair at ');
          write_time; writeln;
        end;
  end;

{ Conditional events }
procedure start_removal;  { C1 }
  begin
    while (mechanic.num_avail>0) and (not empty (q1)) do
      begin
        with mechanic do
          log_histogram (mechanic_util,number-num_avail,1);
        acquire (mechanic,1);
        cause (2,take_top(q1),normal(5,1,removal_time_seed));
      end;
  end;

procedure start_repair;  { C2 }
  begin
    while (equipment.num_avail>0) and (not empty (q2)) do
      begin
        with equipment do
          log_histogram (equipment_util,number-num_avail,1);
        acquire (equipment,1);
        cause (3,take_top (q2),normal (8,2,repair_time_seed));
      end;
  end;

procedure run (duration :real; max_C :integer);
  var
    c :cardinal;
  begin
    running := true;
    repeat
      if calendar=calendar^.next then running := false
        else
          begin
            tim := calendar^.next^.item^.time;
            if duration<tim then running := false
              else
                begin
                  while (calendar<>calendar^.next) and
                        (tim =calendar^.next^.item^.time)  do
                    begin
                      calendar_top;
                      case current^.next_B of
                        0: ;
                        1: end_working;
                        2: end_removal;
                        3: end_repair;
                      end;
                    end;
                  for c := 1 to max_C do
                    case c of
                      1: start_removal;
                      2: start_repair;
                    end;
                end;
          end
    until not running;
  end { run };
```

```
procedure initialize;
  var
    i :cardinal;
  begin
    make_sim;
    make_streams;
    make_bin (mechanic,amount_of_mechanics);
    make_bin (equipment,amount_of_equipments);
    make_queue (q1); make_queue (q2);
    { set a shift working }
    at_work := true;
    { set breakdown times for all machines }
    for i := 1 to amount_of_machines do
      cause (1,new_entity (1,i),
             negexp(inter_break_down_time,removal_time_seed));
    { establish histograms }
    with broken do
      begin
        make_histogram (content,1,1,false);
        number := 0;
      end;
    make_histogram (mechanic_util,1,1,false);
    make_histogram (equipment_util,1,1,false);
  end { initialize };

procedure report;
  begin
    writeln;
    writeln ('Number of Machines Broken Throughout Shift');
    print_histogram (output,broken.content,60);
    writeln;
    writeln ('Utilization of Mechanics');
    print_histogram (output,mechanic_util,60);
    writeln;
    writeln ('Utilization of Equipment');
    print_histogram (output,equipment_util,60);
  end { report };

begin
  initialize;
  run (8*60,2);  { close down the shift }
  at_work := false;
  { run the simulation out, ie. do all repairs left }
  run (10000,2);  { 10000 is arbitary large number }
  report;
end.
```

7 Modelling Techniques

Both the hospital and repair systems, as presented in preceding chapters, have a number of simplifying features which are unlikely to be present in any simulation of reasonable complexity. This chapter is concerned with various techniques for modelling more complex systems.

7.1 MODELLING PROBLEMS

In each of the examples discussed in previous chapters, there was only one entity taking part in each activity and most of the activities were constrained by resources. Sometimes, however, both or all the types of object that take part in an activity must have attributes. This may be because they have to be identified individually or because they have to take part in other activities, independently of each other. Hence, active entities must take the place of passive resources. In the repair shop system, for example, the mechanics might engage in an activity which is independent of the machines, such as having a cup of coffee or going to lunch. They then have to be described as entities.

Although entities have attribute numbers and class numbers, they often need other distinguishing characteristics or attributes; the treatment of a patient in the hospital system, for example, might be influenced by attributes such as age, weight, and medical history. Typically, entities from different classes have different types of attributes.

All the queue priorities in both the hospital and repair shop were 'first-in-first-out' (sometimes called first-come-first-served, often denoted by the mnemonic FIFO). While FIFO queues are very common, many systems involve much more complex queue priorities, sometimes based on entity-attribute values. For example, in the hospital system it would be more realistic to assume that the order in which operations are performed is dependent upon the type of operation required.

Whereas the activities that entities undertake are predetermined in both examples used here, they can, in many systems be subject to various uncertainties. Chapter 2 showed that following the completion of an

activity, entities might have a choice of branches to different activities. The choice might be dependent on attribute values, the state of the system, or might be completely random.

7.2 CO-OPERATING ENTITIES

In structuring a simulation, decisions have to be made as to which objects should be entities and which should be resources. Where all items are identical and are used and released by other objects, they can usually be described as resources. This is true, for example, of the beds in the hospital system. A class of objects must be described as entities rather than resources if:

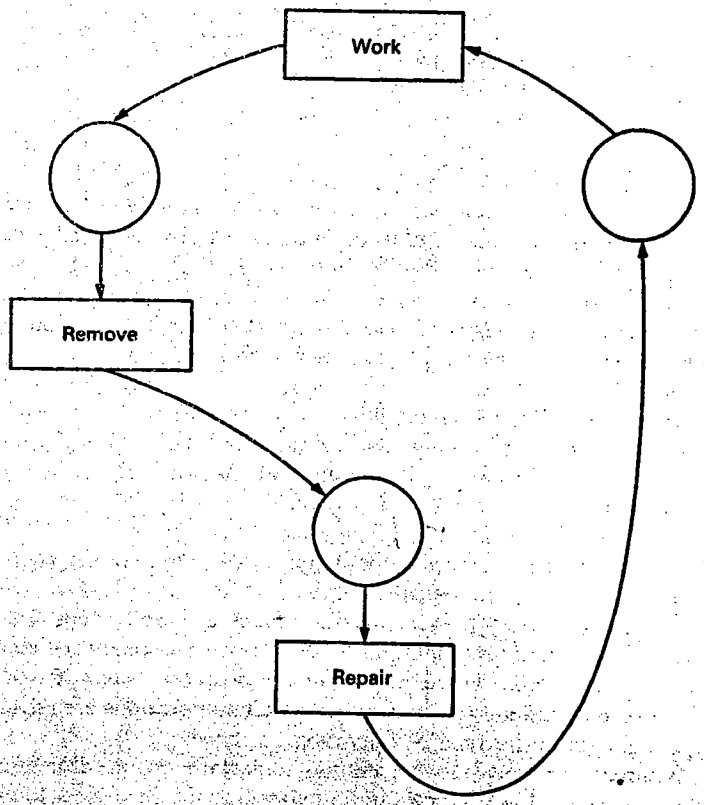

Figure 7.1 Activity-cycle diagram for machine

(*a*) items have to be identified individually or have attributes, or
(*b*) they are the main items taking part in other, independent activities so that they need to carry the activity time on their entity clocks.

Where two such types of entities are mutually dependent on each other in order to take part in the same activity, they are said to be *co-operating* in that activity. Each type of entity has its own cycle or flow of activities which together make up the whole activity diagram.

For example, if the mechanics in the repair shop take time off after each repair to have a cup of coffee, then they are the main entities in the activity of having a cup of coffee. Thus, they must be considered as entities which co-operate with machine entities in *repair* and *removal*. Figure 7.1 shows the activities of the machines, Fig. 7.2 shows the activities of the mechanics, and Fig. 7.3 shows the combined activity-cycle diagram.

Now that the mechanics have individual attributes, their skills can be matched to the different machines and we shall show in the next section

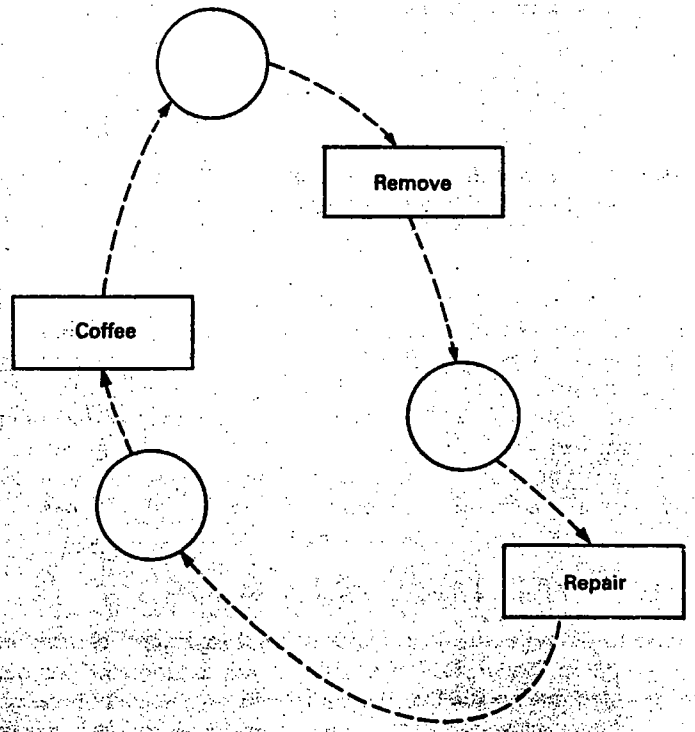

Figure 7.2 Activity-cycle diagram for mechanic

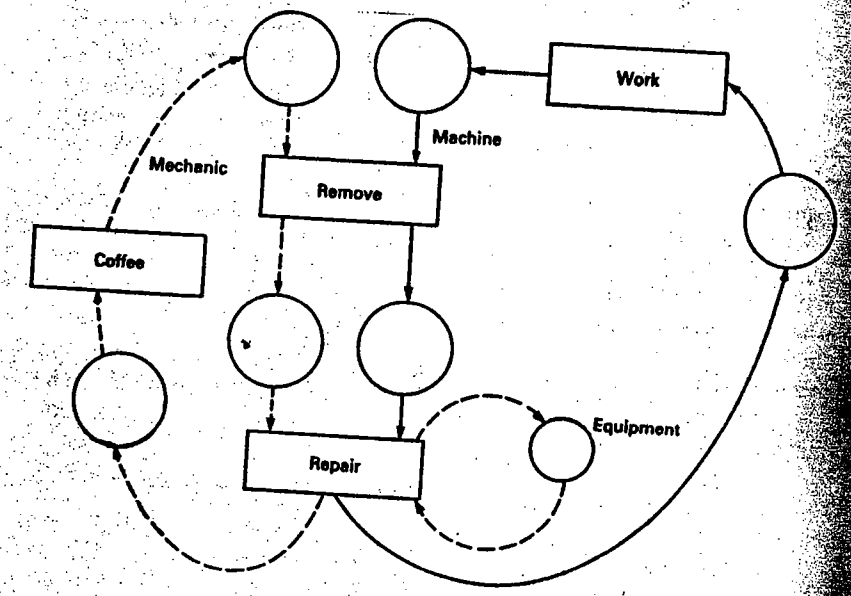

Figure 7.3 Combined activity-cycle diagram of repair shop showing co-operating mechanics and machines together with equipment which is still described as a resource

that they can be given other characteristics. However, the modeller must guard against making the model more complex than it need be.

When two entities co-operate in an activity, they must both be put in the calendar in the conditional event that models the start of the activity. This will be the appropriate C event in the three-phase approach but will be distributed over several scheduled events in the event method.

If both are put in the calendar, a problem arises at the end of the co-operative activity, because each entity in the calendar is linked to a separate and independent bound event. However, as there is only one activity, we want the co-operating entities to cause only *one* bound event and not *two*. There are two ways of solving this problem as follows.

(*a*) One of the entities can be scheduled to enter a *dummy bound event*. In Pascal_SIM, the bound event numbered 0 is a dummy event. This means that the entity is lost at the end of the activity and cannot be put on a queue or engaged to another activity.

(*b*) If access to both entities is necessary, then the solution is to take the second entity off the calendar in the bound event caused by the first entity. In this bound event, the second entity is found at the top of the calendar because the two entities will have entered the calendar together and carried the same time on their clocks. It is not possible for them to have got separated. Thus, in Pascal_SIM the entity can be removed and accessed by:

```
var
  e :entity;
e := take top (calendar);
```

where e is the second co-operating entity. The entity *e* is then available to join queues or engage in other activities.

In the repair shop example in which the mechanics are entities, the second approach is necessary because both the mechanics and the machines must be available to continue their own life-cycle of activities after the *repair* activity.

In the hospital system, the operating theatre could be described as a co-operating entity. Readers should try modelling it in this way themselves.

7.3 ENTITY ATTRIBUTES

In many systems, entity characteristics influence the length and choice of activities. These attributes must therefore be described in the model.

7.3.1 Individual entity attributes

An attribute might be an integer or real continuous variable (for example, patient height, weight, or age in years), an enumerated variable (for example, eye colour), or a relatively complex collection of data (for example, medical history). Within a class of entities, each entity will have a separate value for each particular attribute. The value of each attribute may be fixed for the duration of the simulation, or may change as the simulation progresses.

In Pascal, declared types, such as the entity record, cannot be altered at run time. Thus to extend the entity type, and include attributes in Pascal_SIM, the programmer must alter the basic entity record, and recompile Pascal_SIM. For example, on adding the entity attributes of 'height' and 'weight' to the patient entities in the hospital simulation, the entity record becomes:

```
type
  an_entity = packed record
                avail          :boolean;
                class          :class_num;
                attr, next_B   :cardinal;
                time           :real;
                height, weight :real;
              end;
```

The new_entity procedure must be altered to give *height* and *weight* their initial values and, if Pascal_SIM is a pre-compiled module, all the Pascal_SIM routines must be recompiled for the new entity type. Pascal_SIM is thus tailored to the simulation program. Any entities which are declared are then automatically given attributes. For example, if we want an entity called *temp3* declared as:

```
var
  temp3 :entity;
```

the associated attributes of *temp3* are simply *temp3^.height* and *temp3^.weight*.

In many simulations, unlike the hospital simulation, there will be more than one entity class, and it is likely that entities in different classes will have different attributes. One solution to this problem is to use variant records, in which a different set of fields is provided for each class. If, for example, there was a class of patient entities with attributes described above (i.e., *height* and *weight*) and a class of operating theatre entities with the attribute 'room size', then a variant entity type could be defined as:

```
type
  an_entity = packed record
                avail          :boolean;
                attr, next_B   :cardinal;
                time           :real;
                case class :class_num of
                  1: (height, weight :real);
                  2: (room_size      :real);
              end;
```

where the patient class is class 1, and the operating theatres are class 2. The field class now identifies the appropriate part of the variant record. Appendix 7.A shows how both new_entity and dis_entity must be rewritten using the variant versions of Pascal's *new* and *dispose* procedures.

Although variant records are convenient and neat, they can be dangerous to work with because Pascal run-time systems provide little error-checking to ensure that you are dealing with the right record variant. It is advisable during development, therefore, to check that the entity whose attributes are about to be accessed is of the right class.

7.3.2 Class attributes

An entire class of entities may have attributes that differentiate it from other classes. These attributes are called *class attributes*. Consider an extension of the repair shop example in which there are two classes of machine with different distributions of repair times. The repair-time distribution parameters can now be considered to be class attributes. They can be given a type thus:

```
type
  class_attribute :record
                     mean,sd :real;
                     s       :stream_num;
                   end;
```

where the repair distributions for each class will both be Normal, but with different means and standard deviations. It is useful to make the stream number a class attribute so that the activity times for the different classes can be sampled from different streams (see Chapter 8).

Each entity has a class number which can be used to relate the entity to its class attributes:

```
var
  class_attr :array [1..max_class_num] of class_attribute;
```

We can, for example, find the mean activity time of the current entity as follows:

```
class_attr [current^.class].mean
```

7.3.3 Using sets to store Boolean attributes

In some simulations, control of the logic is dependent upon relative values, for example, 'higher than 3 metres', or previous entity actions, for example, 'operation completed'. Each of these attributes is of a Boolean type, since it can only have the value true or false.

Obviously, Boolean attributes can be added to the entity record in Pascal_SIM, but sometimes it is more efficient to use the Pascal set structure. Members can be added to and deleted from the set as necessary, and membership of the set can be tested with the *in* operator. For instance,

suppose we want to keep a record of all machines that have previously broken down. We can declare a set thus:

```
var
  have_broken :set of amount_of_machines
```

and add a machine to the set

```
have_broken := have_broken+[current^.attr]
```

and test for membership of the set as desired

```
if current^.atter in have_broken then ...
```

7.4 QUEUE PRIORITIES

7.4.1 Simple priorities

Many systems have FIFO queues, but there are a number of other simple queue priorities which are independent of the state and size of the queue, or the entities within the queue. For instance:

(*a*) last-in-first-out (LIFO) is the opposite of FIFO, where entities are removed from the back of the queue first, and
(*b*) in select-in-random-order (SIRO), an entity is selected at random from all those that queue. SIRO can be thought of as corresponding to no queue discipline.

LIFO can be implemented simply by selecting entities from the tail rather than the top of queues when they start an activity. For SIRO, every entity in the queue is numbered, so that one can be sampled and removed from the queue. Chapter 4 explains how to sample from a discrete distribution.

7.4.2 Priority by attribute

In many instances, access to service by activities is dependent upon entity attributes. For instance, in the hospital example, there may be an admissions policy whereby patients requiring admission are ordered by attributes such as age, illness, etc. Therefore, it is necessary to search the appropriate queue to extract the entity with the highest priority.

In Pascal-SIM, this involves moving a pointer through the entire queue, and investigating the attribute of each entity. Whatever that attribute, the search algorithm is the same:


```
< move to top entity in queue >;
present := < some initial value of attribute >;
while < not at end of queue > do
  begin
    if < value of attribute for this entity is better
       than present > then
    begin
      present := < this value >;
      < remember this entity >;
    end;
    < move onto next entity in queue >;
  end;
< take the remembered entity >;
```

For priority rules which involve a number of attributes, the same iteration applies, but the test on the attributes is more complex.

In some instances, entities may have to fulfil certain minimal requirements, in which case a search of the queue may either find no suitable entity in the queue or else may find several. In the later case, it will be possible to stop the search after finding the first suitable entity. The requirements may change systematically or randomly as the simulation progresses.

In the hospital example, suppose the entity record has been extended by an age field (of type *real*) and we wish to give priority to older people for admission to hospital. Rather than take the top entity off the queue, e.g., **take_top** (*q1*), we have to search for the oldest patient thus:

```
var
  temp,pick   :entity;
  present     :real;
present := 0;
temp := q1^.next;
while temp<>q1 do
  begin
    if temp^.age>present then
      begin
        present := temp^.age;
        pick := temp;
      end;
    temp := temp^.next;
  end;
```

we can now remove the oldest patient with take (*q1*,*pick*);

It is good programming style to establish code concerning queue priorities as a function or procedure. Within a simulation model, it can

then be used by a number of separate conditional events. This is particularly important in the event method where the conditions are dispersed amongst the scheduled events. Furthermore, the modules, if required, can be moved between simulation programs.

7.4.3 Priority as single measure

Some simulation programming languages (including GPSS, which is discussed in Chapter 12) treat priority as a single numerical attribute. Thus within conditional events, the entity with the highest value priority is taken off the queue. In other parts of the simulation program, the programmer must arrange for priority to be assigned and altered as necessary.

Where some classes of entities always take priority over others, but share the same queues (perhaps, for example, emergency and walk-in patients in a hospital), each entity can be assigned a numeric priority depending upon its class. In Pascal_SIM, an implementation of the take routine could be developed that simply removes the entity with the highest priority from the queue, using FIFO to choose between entities with equal priority.

This method of dealing with priorities has the advantage of simplicity. Furthermore in some systems, such as the ordering of jobs in computer systems, this is an accurate reflection of the way in which priorities are allocated to queues. However, where priorities are complex, completely random, or dependent on the state of the system, this simple approach can be extremely limiting.

7.4.4 Activity priorities

Chapter 2 showed that the ordering of the list of conditional events in the executive of the three-phase approach determines the priorities given to these events in the simulation. Alternatively, in the event method, the order in which conditions are attempted within the scheduled events determines priorities.

In the hospital simulation, for example, *hospital-stay-only* patients receive priority for admission over *operation* patients, because the conditional event that engages these patients to a bed is attempted prior to the conditional event for operation patients.

The ordering of the conditional events, in the three-phase approach, thus provides a simple and effective means of giving priorities to activities. However, problems arise when there is a lack of independence between conditional events such that one has to be recalled from within another. For example, if the performance of C1 requires a resource X and C2 requires a resource Y, but frees some of X, then C1 will have to be reattempted if C2 is successful. This is further complicated when there is mutual dependency where, for example, the performance of C1 requires X but frees Y, and C2 requires Y but frees X (i.e., they swop resources). In the three-phase approach, these are best dealt with by putting the interdependent conditional events into one long, and rather untidy, conditional event. The logic for mutually exclusive events where there is resource swopping is as follows:

```
while < there is a queue for C1 and a queue for C2 > do
  begin
    < take an entity from the queue for C1 >
    < take an entity from the queue for C2 >
    < set the time for the end of activity 1 >
    < set the time for the end of activity 2 >
  end;
while < there is a queue for C1 and X is available > do
  begin
    < take an entity from the queue for C1 >
    < acquire resource X >
    < return resource Y >
    < set the time for the end of activity 1 >
  end;
while < there is a queue for C2 and Y is available > do
  begin
    < take an entity from the queue for C2 >
    < acquire resource Y >
    < return resource X >
    < set the time for the end of activity 2 >
  end;
```

In the event method, similar logic must be incorporated in the appropriate scheduled events.

7.5 BRANCHING

Chapter 2 mentioned different ways in which branching might arise in a simulation. Two of these are now discussed in more detail.

7.5.1 Branching by probability

Where there is branching from activities, the branches are assigned probabilities. The route selected by each entity depends on a random number sample.

For example, suppose that entrants to a bar either find a table to sit at, or go up to the bar to order a drink. By observing the bar in action, it is possible to record the number of entrants, n, where x find a seat, and $n - x$ go to the bar. Thus for each individual entrant, we can assume that the probability of finding a seat is x/n, and the probability of going to the bar is $(n - x)/n$. Bar customers can be routed appropriately in a simulation model by sampling from a uniform (0, 1) distribution in the arrival B event, and given to the appropriate queue. Chapter 4 showed how to sample one branch from any number of branches using the **sample** function.

The hospital simulation can be simplified by the use of branching. Suppose all patients are generated by a single arrival distribution, and of these 30 % require an operation, then this can be modelled as a single arrival B event that routes entities to the appropriate queue thus:

```
procedure patient_arrives; { B1 }
  begin
    if rnd (patient_seed) <= 0·3 then give_tail (q2,current)
      else give_tail (q1,current);
    cause (1,new_entity(1,1),negexp(6,patient_seed));
  end;
```

In this example, a second feeder with a random variable to generate arrivals, is replaced by probabilistic branching. The number of stochastic variables thus remains the same. Where probabilistic branching replaces a simple deterministic decision rule, it will increase the variability of the model. The benefits of increasing the accuracy are thus counterbalanced by the larger variance that is bound to appear in the results.

7.5.2 Branching by attribute

In many systems, the route taken by entities through the activities will depend upon attribute values. For example, in modelling a sea port, we may classify incoming ships by tonnage, such that ships within different tonnage categories will undergo different activities. As with branching by probability, the appropriate tests for deciding which branch to take are performed in the bound event representing the end of the activity from which the entity has branched.

7.6 SUMMARY

Modelling attributes, co-operating entities, priorities, and branching are important techniques to master in writing simulation programs. Pascal_SIM can be adapted to deal with any of these complexities. In providing entities with attributes, Pascal_SIM has to be amended and recompiled for each simulation requiring different entity attributes. Moreover, where these attributes are used to determine queue priorities, different procedures with very similar logic will have to be written for each type of entity with different attributes.

Priorities may be determined by the method of selection from queues, the characteristics of entities, and the ordering of activities. The logic and variability of the simulation will also be determined by the use of branching between activities.

To some extent, use of queue, entity, and activity priority is interchangeable. In many simulation languages and packages, only one of these is available. However, where there is a choice the simulation developer must choose the method that most closely models the particular system under consideration and provides the most flexibility.

7.7 EXERCISES

For some of these exercises you will need to adapt Pascal_SIM in the different ways described in the chapter.

1. *Hospital System.* Carry out the following tasks.
   (*a*) Change the simulation described in Chapter 6, to generate only *one* set of patient arrivals called *patient_arrival* and then branch to $q1$ and $q2$.
   (*b*) Model the theatre as a co-operative entity.
   (*c*) Try changing the waiting list priorities:
      (i) select the patients randomly from the waiting list;
      (ii) give the patients in the waiting lists random priorities (1, 2, or 3) and select them for admission in priority order, giving preference where there is a choice to those who do not require an operation.

2. *Repair Shop System.* Carry out the following tasks.
   (*a*) Adapt the simulation described in Chapter 6, to take into account the time that mechanics spend having a cup of coffee after each repair, assuming that this takes exactly 10 minutes.
   (*b*) Suppose the different mechanics have different experience in machine repair such that:
      (i) one is inexperienced and takes an average of 5 minutes;
      (ii) two take an average of 8 minutes;
      (iii) one is very experienced and takes an average of 11 minutes.
   Adapt the simulation used in (*a*) to take this into account and compare your results with those from (*a*).

3. Plant A makes a series of castings each of which is transported to plant B on a single track railway by a locomotive (the only locomotive in the system). Plant B repairs the faults and paints the castings. They are then transported back to plant A and dispatched to the customer. Two strategies are proposed for the locomotive:

   (*a*) after arrival at plant B, it waits until a casting is ready for the return journey;
   (*b*) if no casting is ready for the return journey but a casting is waiting at plant A, the locomotive makes the journey empty.

   Design a simulation to discover the best strategy:

   (i) by using activity-flow diagrams, and
   (ii) by describing the events in pseudo-code.

   State clearly the assumptions that you are making.

4. In a study of public transport provision, you have been asked to simulate a bus route with a view to changing the frequency of the service.

   (*a*) Describe the buses as entities with the additional attributes: their next stop, capacity, and list of passengers. Describe the passengers as entities with the additional attribute of destination stop. Recompile Pascal_SIM to provide for these additional attributes.
   (*b*) Draw an activity diagram and write the pseudo-code for the events.
   (*c*) Write this simulation in Pascal_SIM and test it with the following data:

   (i) there are two buses which describe a full cycle of 10 stops and each has a capacity of 30 seats;
   (ii) the time taken for buses to move from one stop to the next is Normally distributed with a mean of 5 minutes and a standard deviation of 1·5 minutes;
   (iii) passengers arrive randomly at each stop at a rate of 1 every two minutes; of these 30 % get off at the next stop , and 50 %, 15 %, and 5 % at each of the subsequent stops, respectively.

   (*d*) Use data from your own local bus route. Print out information about the queues at the bus stops. State any assumptions that you are making.

5. Two trucks with the same weight capacity move a pile of assorted loads of different weights. The items are loaded by crane until a truck is full. The truck then moves them to a second pile, unloading with a second crane and returning empty.

   Write a simulation to determine how long it takes to move the pile and how many journeys are made. Use this simulation to determine if it is quicker to use one large truck with twice the carrying capacity. State any assumptions that you have to make.

7.A EXTENSION OF ENTITY RECORD TO USE ATTRIBUTES

To use variant entity records, both **new_entity** and **dis_entity** must be rewritten. For example, with two entity classes in the simulation, each with a different variant, **new_entity** and **dis_entity** should be:

```
function new_entity (c :class_num; a :cardinal) :entity;
  var
    e :entity;
  begin
    case c of
      1: new (e,1);
      2: new (e,2);
    end;
    with e^ do
      begin
        time := 0.0; avail := true;
        class := c; col := nul;
        attr := a; next_B := 0;
      end;
    new_entity := e;
  end { new entity };

procedure dis_entity (e :entity);
  begin
    if not e^.avail then sim_error ('disposing of a busy entity')
      else
        case e^.class of
          1: dispose (e,1);
          2: dispose (e,2);
        end;
  end { dispose entity };
```

8

Validation and Experimentation Techniques

A model must be tested to ensure that it is reliable, error-free, and has credibility with those who are to use it. Once is has been fully tested, factors, particularly the decision variables, are varied to determine the behavior of the model under different circumstances and to find the estimates of various parameters. This process is called *experimentation*.

In Chapter 5 we showed that several replications are generally needed in order to estimate the means of response variables and the variability of those estimates. Although computer time is much less expensive than it used to be, it is clearly very tedious to have to do many runs in order to arrive at one set of results. This chapter will describe techniques which are available to reduce the number of runs needed for a required accuracy of results.

8.1 VERIFICATION

We shall define *verification* to mean the tasks associated with checking the model and corresponding programs to ascertain that they perform as intended.

A program should be built up in stages that are likely to be as error-free as possible and can be easily tested. The best way of doing this is to use a *top-down* approach in which the programmer provides the overall structure and then fills in the details. This approach can be adapted to the overall process of developing simulations. We recommend that those writing simulation programs should specify a model (as explained in Chapter 6) and then build up the program in stages to meet the specification, in the following way:

(*a*) Code the events, the executive, and the initialization module as simply as possible, using constants rather than variable parameters and simple parametric distributions such as negative exponential and uniform distributions. Leave out any scheduling or more complex matching of entities at this stage and get the program running. It is helpful to print the time and event names to the screen and record the attribute numbers of the entities starting and leaving activities. This provides useful information to indicate whether the program is running as expected.

(*b*) If it is a steady-state simulation, perform a steady-state analysis and collect the results in histograms to check that they look sensible. This is the stage reached with the programs described in Chapter 6.

(*c*) Change the program so that decision variables can be read in from the keyboard and check that it produces the same results as in (*b*).

(*d*) Change the simulation to sample from distributions based on results from data-collection activities and perform further steady-state analyses.

(*e*) If necessary, introduce more complicated logical processes and entity characteristics (see Chapter 7), one at a time, and test at each stage. The logical processes should be coded as procedures or functions so that they can be used and updated separately.

(*f*) Run the program using a wide range of different data and different stream numbers in order to detect more elusive errors.

Pascal_SIM is provided with approximately 30 error traps to pick up many of the errors a programmer is likely to make. Some implementations of Pascal also provide dynamic debugging facilities which are extremely useful for tracing obscure errors such as the use of wrong variant records (see Chapter 7).

8.2 VALIDATION

Validation is the process of checking the model, rather than program logic. We find out whether the model looks and behaves, in important respects, like the real system. If it does not, then assuming that the verification activities have been done properly, we must go back to:

(*a*) the logic of the model structure and review the assumptions, and
(*b*) the data collection and analysis to ensure there are no serious errors or hidden assumptions.

The cyclical process of making assumptions, building a model, validating it, and starting again should be present in all model-building exercises. However, it is particularly important in simulation work because models are frequently complex and though the individual parts may seem quite

reasonable in themselves, when put together they may poorly reflect the behavior of the system as a whole. There are two main approaches to validation which are:

(*a*) to get the users to check that the simulation is running as it should be;
(*b*) to get statistical evidence that the simulation produces similar results to the real system.

8.2.1 Checking by users

Those for whom the simulation is designed, whether management staff, doctors, or union representatives, cannot be expected to understand fully the simulation coding. However, they should be able to understand an activity diagram and should be involved in determining the objectives and hence the logic and the level of detail of the simulation.

A display of simulation results while the program is running, showing the build up of queues, the states of entities, and the use of resources enables the user to see if the simulation looks and feels like the real system. Visual simulations, which are particularly good for this purpose, are described in Chapter 9.

8.2.2 Use of statistics

Once a program is running without errors, the next step is to see how results compare to those produced by the real system. This necessitates additional data-collection work. Data about the 'output' variables such as average queue lengths, server busy time and average waiting times will be needed to compare with those produced by the simulation.

The standard technique for validating predictive models is to run them from some time in the past up to the present time. However, this can only be done if appropriate past data is available. Although this technique can be very instructive, it can also be misleading. It is quite likely, for example, that policies or resource availability have changed in the period under study. If so, the results can only be accurate if these changes are also taken into account. Statistical tests can help establish whether there is a significant difference between the results from the simulation and the information collected from the system relating to the present point in time.

8.2.3 Dealing with discrepancies

The reason for a significant discrepancy between the simulation results and measures taken from the real system must be determined. It should be reasonably easy to check whether errors have crept into the data collection or analysis.

However, problems that arise with the model structure and assumptions may be more obscure. A common problem is to find that resource use in the real system is high and queue length is low compared with the simulation results because the system adapts in various ways to reduce the undesirable effects of long queues and idle servers: for example;

(*a*) where a substantial number of people fail to join queues when they are long (see Chapter 11),
(*b*) an appointments system, or
(*c*) the introduction of a queue of non-urgent jobs which is processed in slack periods.

In the initial simulation design, these details may have been deliberately ignored in order to simplify the model, but once such assumptions are seen to have a significant effect on the results, the simulation structure should be changed to take them into account.

8.3 SENSITIVITY ANALYSIS

In *sensitivity analysis*, the responsiveness of the model to different assumptions and changes in factor levels is tested. However, the number of factors should be kept as small as possible, because there is a geometric increase in the size of the analysis for each factor selected. The analysis is usually performed at two levels for each chosen factor. Factor analysis may be used for testing sensitivity (see Chapter 5).

8.3.1 Sensitivity analysis in validation

The reasons for using sensitivity analysis as part of the validation process are as follows.

(*a*) To ascertain that the model still produces sensible and credible results when factors and assumptions are varied.
(*b*) To look for ways to simplify the model structure; if a change in factor level in a certain range has no effect on model output, then it may be possible to leave out certain resources or activities. If, for example, we included the activities of the doctors in the hospital example, and varied their number over a sensible range, we might find that it had no effect on patient throughput. We might then decide to omit them and their activities from the model.
(*c*) To check the effects of using parameters which are based on

inadequate information where data relating to certain factors is poor or non-existent. For example, it is very difficult to get reliable information about the time that working people spend on their different activities. However, the simulation can be run using the upper and lower limits of the times of these activities. This will show the effects of the uncertainty and indicate the time and effort to be spent on getting good data.

It is almost impossible, of course, to use sensitivity analysis for all of these purposes at once, because they are likely to interact. Therefore, sensitivity analyses must be carefully selected and planned.

8.3.2 Sensitivity analysis in experimentation

Sensitivity analysis is also used as part of the *experimentation* process to explore the effect, on the model results, of changing certain factors or the random number streams.

In the hospital example described in Chapter 6, the decision variables are the number of beds and the opening times of the operating theatre. The chosen maximum and minimum number of beds and extremes of operating theatre times would be selected to be realistic values of interest to the management. They might, for example, be:

	Minimum	*Maximum*
Beds	10	30
Theatre opening time	2	8

The simulation would be run four times to find the average queue lengths for each combination of factors.

8.4 VARIANCE REDUCTION

Chapter 5 explained that responses from a simulation run, or part of a simulation run, were samples from probability distributions. Simulation experiments are often aimed at estimating the means of these distributions. The obvious method is to use the sample mean of several replications or independent batches. However, the greater the variance of the response distribution, the greater are the number of samples that will be needed to get an accurate estimate of its mean. Additional replications are likely to be expensive in computer time and effort. Therefore, *variance-reduction* methods have been evolved to reduce the variance of the estimate of the response means.

8.4.1 Replications

The performance of several runs to estimate the response mean is called the *method of replications*. With n replications the average response, μ_x or $E(X)$, is estimated by $\bar{x}$. The improvement in variance made by using n replications, rather than one single run for the estimate, is given by:

$$\text{Var}\,(X) - \text{Var}\,(\bar{X})$$

If the n replications are independent, then

$$\text{Var}\,(\bar{X}) = \frac{\text{Var}\,(X)}{n}$$

The variance reduction is, therefore:

$$\text{Var}\,(X) - \text{Var}\,(\bar{X}) = (1-1/n)\ \text{Var}\,(X)$$

8.4.2 Constant rates

Constant arrival rates or constant lengths of stay can considerably reduce the variance of results. However, such constant values must be used with extreme caution because some of the stochastic characteristics of the system will be lost. Queue lengths will be underestimated and resource use overestimated. A simulation incorporating constant values is useful for some models and may provide lower or upper bounds of the desired results.

8.4.3 Common random number streams

This technique is used for determining the difference between results when the factor levels of decision variables are changed. The results from a run of a steady-state simulation might be mean resource utilization, queue length, or cost (see Chapter 5). If random variable X is the result from a run at one factor level and Y is the result from a run at another factor level, then $\bar{x}-\bar{y}$, the difference between the average of n runs at one factor level and of n runs at another factor level, is used to estimate the mean difference, $E(X-Y)$.

In order to reduce the number of necessary runs we want to minimize the variance of this estimate:

$$\text{Var}\,(\bar{X}-\bar{Y}) = \text{Var}\,(\bar{X}) + \text{Var}\,(\bar{Y}) - 2\ \text{Covar}\,(\bar{X},\bar{Y})$$

If $\bar{X}$ and $\bar{Y}$ are independent, then $\text{Covar}\,(\bar{X},\bar{Y}) = 0$.

If, however, x_i and y_i (resulting from the ith run) are matched pairs

such that the differences between them can be assumed to be due to the changed factor alone, then $\bar{X}$ and $\bar{Y}$ will not be independent. The covariance will be large and thus the variance of $\bar{X}-\bar{Y}$ produced in this way will be very much smaller. This effect is produced in a simulation experiment if the different distributions in the simulation use different random number streams. These are kept constant between the two experiments while one or more factor levels are changed. In the hospital simulation described in Chapter 6, for example, there are different random number streams for each of the following:

(*a*) each interarrival distribution,
(*b*) the length of stay of patients not needing an operation,
(*c*) pre-operative stay,
(*d*) operation length,
(*e*) post-operative stay.

If, for example, the number of beds were to be increased, there would be a resultant decrease in queue lengths and waiting times but, nevertheless, all the arrival rates and activity times would remain the same between the two experiments and the same entities would use the resources in the same order.

If another decision variable, the theatre opening time, were to be introduced, the following would remain the same between the two experiments:

(*a*) the number and timing of the demand for operative and non-operative treatment,
(*b*) the lengths of stay of patients not needing an operation and the necessary pre-operative and post-operative stay of those having an operation,
(*c*) the order and lengths of the operations.

The same entities would arrive for operations but would have to wait longer in a hospital bed until theatre time became available. Because there would be competition between the two groups of patients for resources, patients from the different groups might well be admitted for treatment in a different order from those in the comparative simulation run. The same entities would thus be given different lengths of stay in the two runs. However, the order of entities within each group would remain the same.

In a simple simulation, if only one factor is changed (as above), then the *differences* between the runs may be largely deterministic rather than stochastic. Although statistical analyses for matched pairs of data may be used, even this is likely to produce very conservative guidelines for deciding on the significance of results. In more complex simulation experiments the effects become less predictable when the following occur.

(*a*) Several factor levels are changed at the same time because they may affect each other unpredictably.
(*b*) There is feedback in the system (see Chapter 11). If, in the above example, patients had shorter lengths of stay if there were a shortage of beds (which is often true in practice), then with the provision of more beds, many patients would stay longer. Then there would no longer be an exact match between the lengths of stay in both experiments.
(*c*) The queues are not FIFO and the entities are selected off the queues in a different order in the two experiments, thus there would no longer be a match between entities.

A different problem arises in attempting to repeat comparative runs with independent sets of random numbers. Suppose that the first comparative runs, in the above example, use random number streams 1–5 to give the values of x_1 and y_1. Then an independent estimate of the effect of varying certain factors will need five completely different random number streams. We cannot repeat any of them, even to use them for different distributions, because it may cause correlation between the results. Therefore, a simulation of any size and complexity will need a large number of different random number streams.

In summary, the use of different random number streams for different distributions, and keeping streams common between different experimental runs, is a very effective means of reducing variances in comparative simulation runs, but it makes a statistical analysis difficult to interpret and may require very many random streams.

8.4.4 Antithetic variables

If there is a stream of pseudo-random numbers $u_1, u_2, u_3, u_4 \ldots$, then we have $(1-u_1), (1-u_2), (1-u_3), (1-u_4), \ldots$ which is also a stream of pseudo-random numbers. These two sets of random numbers are called *antithetic variables*. Either could equally well be used in a simulation.

The method of using antithetic variables is based on the supposition that if a random number stream produces a high result, then the opposite random number stream is likely to produce a correspondingly low result. If a simulation is run with two random number streams having antithetic variables, the average results from those runs will be much nearer to the mean than the average of results from two runs using independent random number streams.

In a simple model this is certainly likely to be true. For example, in a simple queueing model such as the first hospital system described in Chapter 2, if the interarrival distribution is sampled from a random number stream which has a large number of high values, then the number of arrivals will be low and the average queue length will be correspondingly short. If the simulation is run again with the opposite random number

stream, then the number of arrivals will be high and the average queue length will be long. An average of the two values should provide a value closer to the expected queue length than either one of them.

If we add another activity to this simple queueing system, then the output from the first activity becomes the arrival rate for the second. Thus the rate of arrivals in the system at any point in time will eventually affect every activity, and for later activities this effect will be considerably delayed by the service times. The effects of antithetic variables will be further confused by branching and feedback in the model.

This method works best for simulations with common random streams but, even so, the larger and more complex a simulation is, the smaller will be the negative correlation between runs using antithetic variables. Thus it is most effective for small and relatively simple simulations.

This technique can be used with Pascal_SIM if a second set of random number streams are introduced such that they are antithetic to the present 32. A Boolean variable called *antithetic* is declared and when it is *false* the random numbers produced by **rnd** are the normal ones and when it is *true*, each is subtracted from the value one. The additional coding to provide this variance-reduction facility is thus the declaration of one global variable and an extra line in the procedure **rnd**.

Antithetic variables were added to the program of the repair shop example described in Chapter 6. They showed little benefit over using independent random number streams. In 20 runs, the correlation between results were as follows.

Mean number of machines broken throughout shift:	−0·623
Mean utilization of mechanics :	−0·438
Mean utilization of equipment :	−0·388

These are negative, as one would expect, but not very large. Calculations of means of several runs, with antithetic variables, were even less encouraging and appeared to provide very little more accuracy than average values calculated over the same number of runs without antithetic variables.

Therefore, in conclusion, this method of variance reduction is intuitively appealing and may be useful for simple simulations but is unlikely to be much value for complex problems. However, as it is so easy to implement, its efficiency for any particular problem can be readily tested.

8.4.5 Control variates

This method is, perhaps, the most commonly described in text books, but is the least understood and used of all methods. Although the formulae always *look* complicated, however, the concepts are quite simple.

If we take the simple health system queueing problem, the average arrival rate is an *input variable* and the queue length and resource use are *output variables*. However, the *actual* arrival rate used in the simulation depends on the random numbers sampled in that particular run. If the actual arrival rate is higher than the theoretical rate, then the average queue length and average resource use are also likely to be raised, and vice-versa. The arrival rate is thus correlated with queue length and resource use.

In the *method of control variates* the difference between the measured factor values (such as the measured arrival rate) and the known average values (such as the input arrival rate) is used to adjust the estimates of the output values (e.g., queue length or resource use). Let us suppose that we are using the control variate (such as the arrival rate), C, to adjust estimates of μ_x, the average of response variable, x, then:

(*a*) the control variate average, μ_c, is known,
(*b*) the measured sample average of the control variate, averaged over n replications (or batches) is $\bar{c}$,
(*c*) the measured average response is $\bar{x}$ averaged over the same replications (or batches).

The problem is to find a better estimate of μ_x than $\bar{x}$. Let us call this estimate $\hat{x}$ where:

$$\hat{x} = \bar{x} + k(\bar{c} - \mu_c) \tag{8.1}$$

Regardless of the value of k, this is an unbiased estimate of μ_x because $E(\bar{c})$ is equal to μ_c.

The key to the successful use of control variates is the choice of k which must be must be chosen to make Var $(\hat{X})$ as small as possible. Taking variances of (8.1):

$$\text{Var}\,(\hat{X}) = \text{Var}\,(\bar{X}) + k^2\text{Var}\,(\bar{C}) - 2k\ \text{Covar}\,(\bar{C},\bar{X}) \tag{8.2}$$

In order to improve on the method of replications, Var $(\hat{X})$ must be smaller than Var $(\bar{X})$ and therefore k multiplied by the covariance must be positive, thus:

if the covariance is positive then $k > 0$
if the covariance is negative then $k < 0$

Var $(\hat{X})$ is minimized with respect to k by setting:

$$\frac{d(\text{Var}\,(\hat{X}))}{dk} = 2k\ \text{Var}\,(\bar{C}) - 2\ \text{Covar}\,(\bar{C},\bar{X}) = 0$$

$$k = \frac{\text{Covar}\,(\bar{C},\bar{X})}{\text{Var}\,(\bar{C})} \tag{8.3}$$

$$k = \varrho \frac{\sqrt{(\text{Var}\,(\bar{C})\,\text{Var}\,(\bar{X}))}}{\text{Var}\,(\bar{C})}$$

$$= \varrho \frac{SD(\bar{X})}{SD(\bar{C})} \qquad \rho = \frac{\text{Cov}(\bar{X}, \bar{C})}{SD(\bar{C})\,SD(\bar{X})}$$

where ϱ is the correlation between $\bar{C}$ and $\bar{X}$.
From (8.2) and (8.3):

$$\text{Var}\,(\hat{X}) = \text{Var}\,(\bar{X}) + \frac{(\text{Covar}\,(\bar{C},\bar{X}))^2}{\text{Var}\,(\bar{C})} - \frac{2(\text{Covar}\,(\bar{C},\bar{X}))^2}{\text{Var}\,(\bar{C})}$$

The variance reduction is therefore:

$$\text{Var}\,(\bar{X}) - \text{Var}\,(\hat{X}) = \frac{(\text{Covar}\,(\bar{C},\bar{X}))^2}{\text{Var}\,(\bar{C})}$$

$$= \varrho^2\,\text{Var}\,(\bar{X})$$

In order to determine an approximate value of k, the standard deviation and correlation coefficient of $\bar{C}$ and $\bar{X}$ can be estimated from several replications of n runs.

The variance of the estimate of a response can be further improved by the use of additional control variates. For example, the average queue length may be correlated with the arrival rate and with the average length of service time in the activity immediately following the queue. If three control variates, C_1, C_2, and C_3 are independent, then the response is estimated by:

$$\hat{x} = \bar{x} - k_2(\bar{c}_1 - \mu_{c_1}) - k_2(\bar{c}_2 - \mu_{c_2}) - k_3(\bar{c}_3 - \mu_{c_3})$$

If ϱ_i is the correlation between X and C_i, then the minimum variance of the estimate is given by:

$$k_i = \varrho_i \frac{SD(\bar{X})}{SD(\bar{C}_i)}$$

In choosing the control factors it is helpful to remember that activity-time distributions will influence queue lengths and also the resource use in related activities. However, arrival rates will only directly influence the first activities in the simulation and will indirectly influence all the activities in the simulation after varying time lapses. Therefore, the following stages are necessary in order to use the method of control variates:

(*a*) identify any factors that might be correlated with the output variables and adjust the simulation to collect and average the sample values for those factors,
(*b*) estimate the variance of each factor mean and the correlation between it and the output variables it is associated with,
(*c*) adjust the estimated expected values of the output variable by the weighted control variates.

The major problem in implementing the method of control variates is that there is a considerable amount of work in estimating each value of k. If the user wants to base results on n replications, then several independent sets of n replications must be made to estimate k accurately. This is only likely to be worthwhile if the user has a great deal of experimentation to do.

If the user wishes to base the results obtained on one run rather than n replications, the expression for k simplifies to:

$$k = \varrho \frac{SD\,(X)}{SD\,(C)}$$

where ϱ is the correlation between C and X. The variance reduction becomes $\varrho^2\,\text{Var}\,(X)$. This can be compared to the benefit of using n replications (see section 8.4.1) where the variance reduction is

$$(1 - 1/n)\,\text{Var}\,(X)$$

If ϱ^2 is greater than $(1 - 1/n)$ then the results will, on average, be better than those using n replications.

Suppose, in the repair shop example, we decide to estimate the average number of broken machines throughout the simulated day based on *one* simulation run. The control variates available for use are: the machine inter-breakdown time, the removal time, and the repair time. Table 8.1 shows their average values and the estimates of k from the correlations and standard deviations produced from 10 simulation runs.

In studying these results it must be remembered that the estimated values of the control variates are subject to sampling error and are different from their theoretical values. Small correlations are particularly dangerous because they may even have the wrong sign. The repair time in Table 8.1, for example, shows a negative correlation with the average number of broken machines which is illogical. Therefore, in order to be confident that the method will reduce the variance, it is best to use the control variates which are based on reasonably large correlation coefficients. In this example, the first two control variates could be used to adjust the estimated average number of broken machines from a single simulation run.

In conclusion, the simulation has to produce more output in order to provide the data relating to the factor values as well as the output values (although these data may well be needed for validation purposes anyway). Once these measures are available, several independent runs will be needed to estimate the standard deviations and correlations between the sample means. For 'one-off' simulations, these runs may be adequate to calculate the output measures with sufficient accuracy without introducing the control variates. However, control variates are extremely useful for ex-

Table 8.1 *Control variates constants calculated from 10 independent runs of repair shop simulation where response variable is average number of broken machines and control variates are: (a) machine inter-breakdown time, (b) removal time, (c) repair time*

Average number broken machines		*Inter-breakdown time*	*Removal time*	*Repair time*
Theoretical mean	–	60	5	8
Sample mean	4·17	62·36	5·02	7·93
Standard deviation	0·42	4·64	0·084	0·46
Correlation with response	–	−0·664	0·797	−0·211
k value	–	−0·0593	3.9300	−0·5528

perimental work because, once calculated, they can be used for many different runs at various factor levels and with changed assumptions. They are particularly useful for simulations, with many variable responses, which are used on a long term basis.

8.5 SUMMARY

Techniques for *verification* and *validation* are time consuming but important if the model is to be credible and error free. The verification of a model is aided by the use of a 'top down' approach. Visual simulations enable users to validate models by ensuring that they reflect the important characteristics of the system being modelled. Statistical validation using hypothesis testing may also be helpful.

Sensitivity analysis, showing the effects of and interactions between different factors, both tests the model and provides a starting point for experimentation. Further experimental work, using some or all of the techniques discussed in Chapter 5, will also undoubtedly be needed.

Variance reduction techniques reduce the number of necessary replications in a simulation experiment. The use of *common random number streams* is particularly useful for comparative simulations. *Antithetic random numbers* may be used for simple simulation experiments. The method of *control variates* is particularly helpful when a simulation has to be used on a long term basis to explore the effects of using different data or of changing policies in the organization. Most of the methods discussed become more complex to implement as the simulation becomes larger and the benefits of using them have to be weighed against the extra work involved.

The use of visual output and the provision of interactive simulations have become increasingly important in recent years and these will be discussed in the next two chapters.

8.6 EXERCISES

1. Adapt Pascal_SIM to provide antithetic sampling and show to what extent this reduces variances in the simple hospital system (Exercise 3, Chapter 5) and in the hospital system (Chapter 6).

2. *Bank System* (Appendix 1.A.1). Use the program developed in Exercise 4, Chapter 6, to carry out the following.
   (*a*) Explain how you would validate the bank system and identify the additional data requirements.
   (*b*) Explain which method of variance reduction is appropriate to this problem and implement it.
   (*c*) Show how sensitive the results of the bank system simulation are to the range of hourly arrival rates throughout the day and to the number of tills in use.

3. *Traffic Light System* (Appendix 1.A.2). Use the program developed in Exercise 5, Chapter 6, to carry out the following.
   (*a*) Explain how to use the control-variate method to reduce the variance of the output from the traffic light simulation from one run, using the overall arrival rate as the control variable. Calculate the control constant from 12 replications and use this to estimate the output values from a single run.
   (*b*) Determine the sensitivity of the simulation to the timing of the traffic lights.
   (*c*) If the volume of traffic is predicted to increase by 5 % a year until the bypass is finished in three years time, predict the evening rush hour build up at these lights in two years time.

4. Study the calculation of control variates in section 8.4.5 of this chapter.
   (*a*) Explain whether the standard deviation of any of the control variable responses, SD(C), could have been calculated theoretically and, if so, how.
   (*b*) Prove that three independent control variates (C_1, C_2, C_3) reduce the variance of the estimate of response X by $(\varrho_1^2 + \varrho_2^2 + \varrho_3^2)$ Var (X), where ϱ_i is the correlation between C_i and X.
   A quick calculation will show that, in this example, the squared estimated correlation coefficients add up to more than one. Explain this apparent paradox.

5. The company running the manufacturing cell, described in Exercise 3 of Chapter 6, wish to extend the cell to accomodate twice the number of products. However, they have decided that:

(*a*) the products lost to a cell (because all of the buffers are full), and
(*b*) the mean time to process products through the cell,

should be no worse than they are now. Furthermore, the buffers for each workstation must, for technical considerations, be of equal size. If a new workstation costs £10 000 and a single buffer space costs £3000, is it better to:

(*a*) increase the size of the workstation,
(*b*) build more workstations, or
(*c*) do a combination of both?

9
Visual Output

In previous chapters, once a valid model had been built, input statistics and experiments were used to produce output statistics of various measures. In these experiments, the Pascal simulation is regarded as a 'black box'. However, visual simulations portray the changing activities and use of resources as the simulation progresses.

9.1 NEED FOR VISUAL OUTPUT

There are many situations when it is useful to view the simulation as it proceeds in time:

(*a*) to validate a simulation – it is helpful to look at parts of the model in considerable detail, in order to check that the logic is correct;
(*b*) to see how various elements interact and to identify the causes of problems such as long queues or the underutilization of resources;
(*c*) to interest users and to help them understand and interpret the simulation.

Where interactive facilities are available, a visual simulation enables users to alter decision variables in response to the information on the screen.

9.2 PROVIDING VISUAL OUTPUT

9.2.1 Different approaches

There are a number of different methods for 'visualizing' the running simulation. The simplest approach is to include some trace statements, so that the values of relevant parameters are written after every time beat.

Although this is a useful aid to verification, it can be difficult to follow as the numbers can change very quickly and become unreadable.

Histograms, such as those discussed in Chapter 5, can be displayed as the simulation proceeds, and extreme results outside the Normal range may be highlighted. Whilst displays such as these are very useful they do not provide an insight into the detailed logic of the simulation. Nevertheless, they are an important adjunct to other types of display.

The types of display discussed in this chapter attempt to portray the changing state of the entities, resources, and activities being simulated. They are based on a diagram or picture of the system which changes systematically as the simulation progresses. The simplest concept is to use the activity-flow diagram. Diagrams are animated by changing the color of activity rectangles and queue circles when they are in use. Flashing lines may indicate the movement of entities between states. Whilst this approach has the benefit of being applicable to almost any simulation, it is much more interesting and generally more useful to produce more sophisticated pictures.

Most of the more recent packages employ *dynamic iconic* displays. An entity is represented by an icon which is a small symbol representing a number, letter, or picture. The background picture represents, as realistically as possible, the system being simulated and then the entities move around the picture as the simulation progresses. The effect is similar to that of a video game. Color can be used to enhance both the quality and realism of the picture.

This chapter will show how this approach can be used to display the logic, queue lengths, and resource use of the running simulation.

9.2.2 Visual interactive simulation

The provision of a dynamic iconic display, and its combination with user interaction, is normally called *visual interactive simulation* (VIS). In the USA, where there is less emphasis on interaction, the term *animation* is more common.

In addition to packages which have been designed specifically to take advantage of VIS, such as SEE-WHY (Fiddy *et al.* 1981), the majority of simulation packages now have some VIS facilities. Many packages use high resolution graphics and avoid the problem of hardware compatibility by specifying (or selling) specialized graphics equipment that must be used. Some packages provide standard displays to reduce the amount of necessary programming work. For instance, a queue composed of letters representing entities is automatically managed by the queue manipulation routines. All the programmer has to do is specify where the queue should appear on the screen.

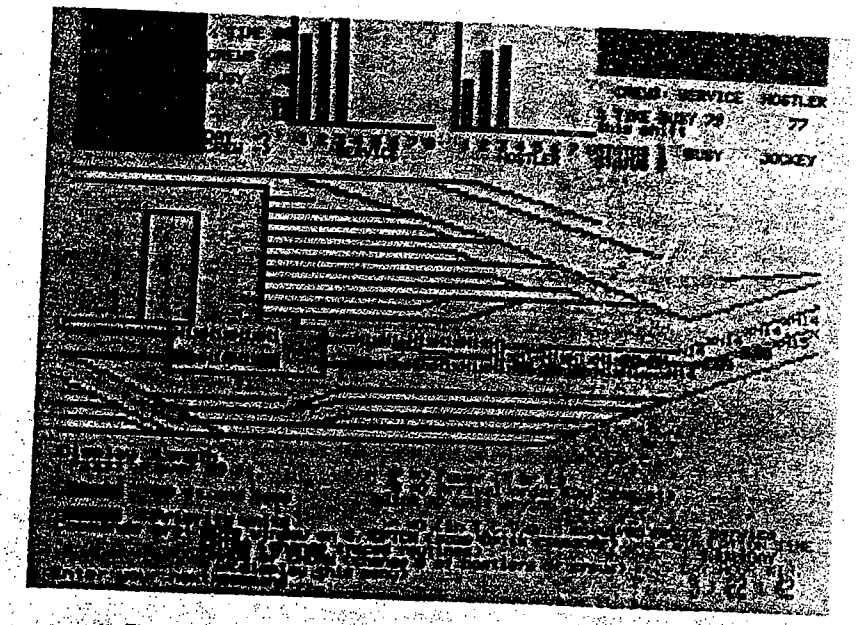

Figure 9.1 Screen display for locomotive service centre model

Figure 9.1 shows an example of the type of quality display that can be produced using a specialized commercial package. This simulation, developed at the University of Western Ontario for Canadian National Railways, uses OPTIK to show a locomotive service centre. The display shows locomotives and repair crews on the available railway lines in the centre, moving around as they complete tasks and start new ones. The user can use zoom facilities to move in on part of the display for a more detailed inspection, or interact with the simulation via the menu option shown at the bottom of the screen. At the top of the screen, there is a histogram of the time that locomotives spend in the yard. The bottom right-hand corner contains the other summary information and the present simulation clock time. This combination of dynamically changing information has made the model easy to comprehend by those responsible for the centre and greatly aids validation (Bell and O'Keefe 1987).

9.2.3 Designing visual output

Designing the 'picture' can take considerable skill. It is important to make sure that it includes all the relevant detail necessary to portray important results from the simulation run. The picture should also be as realistic as

possible so that it is easily understood by people familiar with the system being simulated. The stages in developing the picture are:

(*a*) designing the static background,
(*b*) designing the dynamic component,
(*c*) integrating these with the simulation.

It is advisable to plan the visual output at an early stage in the modelling process as it will help to clarify the structure and output requirements of the simulation program. The static background should show, as realistically as possible, the things that remain constant for the duration of the simulation. This will include the picture or diagram covering the screen and any necessary text explanation. The railway lines in the OPTIK model are, for example, part of the background picture. The background need only be written to the screen at the start of the simulation. The dynamic component of the picture changes with each event. Icons representing entities can be moved across the screen and put into queues. Color can be used to reflect attributes. The visual output must be driven by the simulation as it proceeds. This may be done in one of two ways:

(*a*) the visual output is coded explicitly within the simulation events and executive,
(*b*) instructions are coded automatically by the running simulation and these instructions are decoded by a separate program, either running at the same time or after the simulation run.

This chapter describes how the first approach is implemented in Pascal_SIM. Chapter 14 discusses the benefits and disadvantages of the two methods.

9.3 VISUAL OUTPUT WITH PASCAL_SIM

The visual facilities available in Pascal_SIM are largely hardware independent, and can be implemented for almost any console screen (see Appendix C). They use character facilities rather than specialized graphics capabilities and as a consequence, are, fairly simple. The icons, for example, are letters of the alphabet. Therefore, the quality of the output is considerably poorer than that available in a tool like OPTIK. However, the basic approach and principles are the same.

9.3.1 Combining visual statements with simulation logic

In Pascal_SIM, the facilities for providing visual output are independent of those for driving the simulation logic. Simulations can thus use the same procedures for entity and queue manipulation to provide simulations with or without visual output. The dynamic picture is updated in two ways:

(*a*) when events occur, and
(*b*) after an entire time beat.

The logic for moving around the screen must be written in the events because only there can the current entity and its attributes (such as color) be identified.

For instance, in the hospital system, suppose we choose to portray the queue of operation patients waiting for a bed ($q2$). The bound event *patient2_arrives* should move the arriving entity into the queue thus:

```
procedure patient2_arrives;
  begin
    < take the current entity and add to tail of q2 >;
    < move the current entity towards q2 >;
    < display the new instance of q2 >;
    < create a new entity >;
    < cause the arrival of the new entity >;
  end;
```

In the three-phase approach, the number of resources in use should be written to the screen after the entire time beat. This is because resource levels are artificially depressed during the course of a time beat. For example, the number of beds in use in the hospital system may decrease following a patient discharge, and then increase again in the same time beat due to a patient, who has been waiting in the queue since the previous time beat, being admitted to the bed.

In the event method, it makes little difference whether resource levels are recorded in the events or at the end of the time beat.

9.3.2 Structure of simulation

The structure of a Pascal_SIM simulation with visual output is slightly more complex than that for a non-visual simulation.

The picture must be initialized by giving entities default letters and colors and by programming the static background. The initialization procedure is called **picture**. Code for updating the picture after each time beat is collected in a procedure called **display**. The suggested structure for a visual simulation program is now:

```
Bound events;
Conditional events;
procedure display;
procedure run;
```

```
procedure initialize;
procedure picture;
  begin
    |
    initialize;
    picture;
    run;
    report;
    |
  end.
```

9.4 PROGRAMMING VISUAL OUTPUT

9.4.1 Basic screen control

Pascal_SIM provides all the necessary screen control facilities to run visual simulations. Many versions of Pascal already provide some, or all, of these facilities. Appendix C shows how to alter Pascal_SIM for different consoles and implementations of Pascal.

The procedure **make_screen** initializes the screen to receive control codes, and the procedure **clear_screen** clears the entire screen and homes the cursor to the the top left-hand corner of the screen. This position is taken to be the co-ordinates (1,1). For most screens, which are 24 lines deep and 80 characters wide, the bottom right-hand cursor position is (80,24). The cursor can be moved around the screen to co-ordinates x, y by use of

procedure **gotoxy** (x,y :**cardinal**);

For example

gotoxy (10,1); *write* ('*HOSPITAL SIMULATION*');

will write *HOSPITAL SIMULATION* on the top line of the screen, indented by 10 characters.

9.4.2 Using color

Pascal_SIM assumes the existence of 8 basic colors, which are collected as an enumerated type:

color = (**nul**,**black**,**red**,**green**,**yellow**,**blue**,**magenta**,**cyan**,**white**);

where **nul** means 'no color'. The other eight are the base colors which are commonly provided in many microcomputer systems. Pascal_SIM can,

if necessary, be adapted to provide for the number of colors that are available on a particular machine, where these are more or less than eight, including just black and white (see Appendix C). The foreground color and background color for each character position is set with:

procedure **set_foreground** (c :**color**);

and

procedure **set_background** (c :**color**);

respectively. These procedures do nothing to the screen until there is a *write* statement. All the characters that are subsequently written to the screen are shown in the foreground color on top of the background color. The background color covers only a square the size of the character position, for each character. The effect of the procedures is therefore akin to dipping two brushes in paint but doing no painting until there are further instructions. The routine:

procedure **reset_colors**;

sets both colors back to the screen's default colors, normally white foreground on a black background. The effect of calling the **set_background** and **set_foreground** procedures remains in force until superseded by one of the same routines picking up a different color or else by **reset_colors**.

9.4.3 Static background

The static background can be composed of text and blocks (rectangles) of background color. There is a routine for coloring blocks of text:

procedure **write_block** ($x1$,$y1$,$x2$,$y2$:**cardinal**; b :**color**);

where the top left-hand co-ordinates of the block are ($x1$,$y1$), and the bottom right hand are ($x2$,$y2$). Thus **write_block** (10,10,14,18,**blue**) would produce a blue block with a width of 4 characters and height of 8 characters, whose top left-hand point is situated at (10,10).

Text can be put on the foreground of the block, or entities moved across it, in any color. It is a good idea to use contrasting colors to make the screen legible.

9.4.4 Initialization of entity icons

Every entity class is represented in a class table, which specifies a default letter and color for the entities of that class. The class table is simply an array:

```
class_table :array [class_num] of
              record
                let :char; col :color;
              end;
```

It must be initialized by a call to

procedure **make_class_table**;

Then each entity class that is going to be used in the display is entered in the table using

procedure **enter_class** (*n* :**class_num**; *l* :*char*; *c* :**color**);

where *n* is the class number (see Chapter 3). Entities that are not in the display do not need to be represented in the class table.

In the hospital example, operation patients have class number 2. For example,

enter_class (2,'*O*',*blue*)

will cause these patient entities to be displayed as an *O* and they will be given a default color of *blue*. If the entity attribute **col** is given a different color, then this will overide the default color in the class table.

9.4.5 Dynamic display

Entities represented in the class table can be displayed using:

procedure **write_entity** (*x*,*y* :**cardinal**; *e* :**entity**);

which writes entity *e* at (*x*,*y*), using the letter from the class table.

procedure **write_queue** (*x*,*y* :**cardinal**; *b* :**color**;
q :**queue**; *max_length* :**cardinal**);

writes an entire queue, up to a maximum length of **max_length**, as a horizontal list of letters, with the head of the queue at the right-most point (*x*,*y*). Each entity is displayed individually as a letter in its correct foreground color. If there are more than **max_length** members in the queue, the last entry in the display is an asterisk. The variable *b* denotes the background color of the queue. Thus, for instance,

write_queue (22,12,**white**,*q*1,20);

results in the entities in *q*1 being displayed with the queue head at (22,12), a white background, and the queue stretching to a maximum of 20 places. The movement of entities can be represented on the screen by

procedure **move_v** (*x*,*y*1,*y*2 :**cardinal**; *e* :**entity**; *b* :**color**);

and

procedure **move_h** (*y*,*x*1,*x*2 :**cardinal**; *e* :**entity**; *b* :**color**);

where **move_v** moves entity *e* from (*x*,*y*1) to (*x*,*y*2), where *y*2>*y*1, and **move_h** moves an entity from (*x*1,*y*) to (*x*2,*y*), where *x*2>*x*1. In each case *b* is a background color.

These procedures work by successively copying the letter representing the entity to an adjacent character space, erasing the original, copying it over another space, erasing the new original, etc., until it arrives at the final position. These routines simulate movement on the screen so effectively that the representation of the entity appears to move from one queue to another.

These procedures dictate that the entities 'flow' through the picture from left to right and from top to bottom. Therefore the picture must be designed so that the entities move from queues to activities in this way. This may be difficult if entities have to retrace their steps. The reader may like, as an exercise, to develop new versions of these routines which will move entities in different directions.

When coded and working, the picture may advance too quickly for comfortable viewing. The rate at which the picture changes can be delayed by using the constant **delay_num** and the procedure **delay**; **delay** simply loops for a period determined by **delay_num** and it is used by both **move_h** and **move_v** to delay the 'walking' of an entity across the screen. To delay the simulation after a picture is fully updated in **display**, **delay** can be called a number of times, for example,

for i := 1 *to* 10 *do* **delay**;

The simulation can be made to look more realistic if, rather than providing a blanket delay after every time beat, the simulation is stopped for a delay that is proportional to the time between time beats. This means that large gaps between time beats appear greater than smaller ones. This can be done by recording the time of the last time beat as *old_tim*, and then, within **display**, setting the amount of delay proportional to the difference between the present clock time **tim** and *old_tim*, for instance:

```
procedure display;
  var
    i :cardinal;
  begin
    |
    < finish update >;
    |
    for i := 0 to trunc ((tim-old_tim)/2) do delay;
    old_tim := tim;
    |
  end;
```

Note that immediately after **display** the clock will be updated, thus *old_tim* should be reset to the present clock time **tim** within **display**.

Finally, in almost every picture, it is helpful to show the clock. This can be written to the picture in **display**, before the delay is invoked. If minutes are the basic time unit of the simulation, then the provided procedure **write_time** can be used to convert the clock time **tim** to a days: hours: minutes: seconds display.

9.5 USER INTERACTION

Most commercial VIS packages provide facilities for interaction as well as animation. Those using the simulation for decision-making will normally be the people who are interested in interacting with it. There are two cases where user interaction is very helpful as follows.

(*a*) It may be used when there are decision mechanisms in the system which are too complex, or not well enough understood, to encapsulate in the simulation. For instance, in a manufacturing system, a production controller may make a number of decisions regarding the scheduling of tasks to machines. Rather than trying to capture the decision-making behavior of the controller in the simulation, the simulation can be written so that every time a scheduling decision has to be made, it is passed over to the user. The combination of the simulation plus the controller results (hopefully) in an accurate model of the system. (For an example of this, see Hurrion 1980.)

(*b*) In many systems an overseer or manager can alter parts of the system as required. In the hospital system, for example, an administrator might have the option of opening another hospital ward if the waiting list were very long. Thus we would need to arrange for the user to be able to halt a simulation when the waiting list is seen to be long. Using visual interaction to provide the prompt to the user, such actions can be incorporated in the simulation.

The first of these two types of interaction is often called *model-determined*, the second *user-determined*. Model-determined interaction is quite easy to implement. When a decision needs to be referred to the user, a bound event is scheduled in which there is interaction with the user, followed by any necessary variable changes or scheduling of bound events.

In user-determined interaction, it is essential that the user can halt the simulation, at will, with a routine which polls the keyboard, returning

some pre-determined value if a key has been hit. (In Pascal, it is not possible to just issue a *read* command, since program execution will just wait until a value is entered.) Many Pascal implementations provide a predefined routine that does this. For instance, some UCSD Pascal implementations provide a function called *keypress*, which returns *true* if a key has been pressed, but will return *false* after the value of the key has been read. This can be used in the **display** procedure to initiate interaction, since **display** is called after the end of a time beat, thus;

```
procedure display;
  begin
    < update display >;
    if keypress then
      begin
        < read input >;
        < take appropriate action >;
      end;
  end;
```

A sensible thing to do is to display a number of single-character menu options for the available interactions. Then the user can select a menu option, whilst the simulation is running, which can be picked up after *keypress* returns *true*. User-determined interaction gives rise to several problems as follows:

(*a*) Statistical analysis is invalidated, since the user can alter the value of decision variables at will. For example, if the hospital administrator changes the number of beds in the simulation in one run, he is unlikely to do so at exactly the same place in another run. This invalidates the second run as a replication.
(*b*) Almost certainly, steady-state simulations will be removed from their steady state by any interaction.
(*c*) Where users have the option of resetting resource levels in the middle of a run, the user may wish to reduce resource availability. However, it is difficult to decrease them while they are in use.
(*d*) In many cases, the user is not in a position to give much time to experimenting interactively with a simulation.

Where both interaction and statistical analysis are required, the best approach is to do some interactive runs, recording exactly where and what interaction took place. The timing and effects of the interaction can then be programmed into the simulation and it can be used as an experimental tool in the normal way.

There is a growing body of literature on VIS, some of which deals with these problems, much of which is included in the references here.

9.6 SUMMARY

Visual output is now as important in simulation modelling as the collection of statistical summaries. Using visual output can be a great deal of fun, but can also greatly increase the time taken to program the simulation. For newcomers to simulation, there is a significant danger that the provision of visual output will become an end in itself and the overall objectives of the simulation exercise will be forgotten.

Visual simulations are developed in three stages with a static picture, a dynamic picture, and the integration of the command creating a visual display with the simulation logic. Visual output in Pascal_SIM is quite simple and portable. Displays show entities as letters which, as the simulation progresses, can be moved from one queue, or activity to another. They can also change color as they change state. Visual displays make it possible to introduce interactive facilities. The use of interaction is both powerful and problematical as frequent user interaction will invalidate any statistical analysis.

Chapter 10 shows how visual simulations can be developed for both the hospital and repair shop systems.

9.7 EXERCISES

Exercises 2, 3, and 4 will require the screen control and visual facilities of Pascal_SIM. Readers should look at Appendix C for implementation instructions.

1. Plan the static and dynamic pictures for the following simulations:
   (*a*) the garage (Exercise 2, Chapter 2);
   (*b*) the ante-natal clinic (Exercise 3, Chapter 2);
   (*c*) the manufacturing cell (Exercise 3, Chapter 6);
   (*d*) the locomotive transport of castings (Exercise 3, Chapter 7);
   (*e*) the bus route (Exercise 4, Chapter 7).
   (*f*) the transport of items by truck and crane (Exercise 5, Chapter 7).
2. Implement the screen control and visual procedures of Pascal_SIM and then do the following.
   (*a*) Clear the screen and write 'TEST SCREEN' in white on blue, at the second line from the top on the screen, in the centre.
   (*b*) Draw a block six lines below the message 'TEST SCREEN' in magenta, 11 characters square.
   (*c*) Write the word 'Activity' in red on white, in the middle of the block.
3. Write additional procedures for Pascal_SIM to move entities around the screen from right to left and from bottom to top.

4. Write an animated version of the simple hospital system (using the code from Exercise 6, Chapter 4) as follows.
   (*a*) Plan and program a static picture.
   (*b*) Set up a **class_table** of patient entities.
   (*c*) Represent each entity by a letter and show the patients queueing for admission.
   (*d*) Display a clock in the top right hand corner.
   (*e*) If the picture moves too quickly, delay the simulation so that it can be read.
5. Adapt the simple hospital system in the previous exercise, to stop when a key is pressed and restart when a key is pressed. When this works, adapt this further so that the user can change the number of beds in the ward when the simulation is interrupted. Remember that if the number of beds is to be reduced, you cannot assume that the beds you want to close are unoccupied at the point in time you interrupt the simulation.

10

Hospital and Repair Shop with Visual Output

The previous chapter introduced the idea of visual output, particularly the dynamic iconic display, and the facilities Pascal_SIM provides for programming such displays. In this chapter, we will produce entire programs for the two examples given in Chapter 6, only this time providing visual displays rather than collecting statistics. (At this point, the reader may find

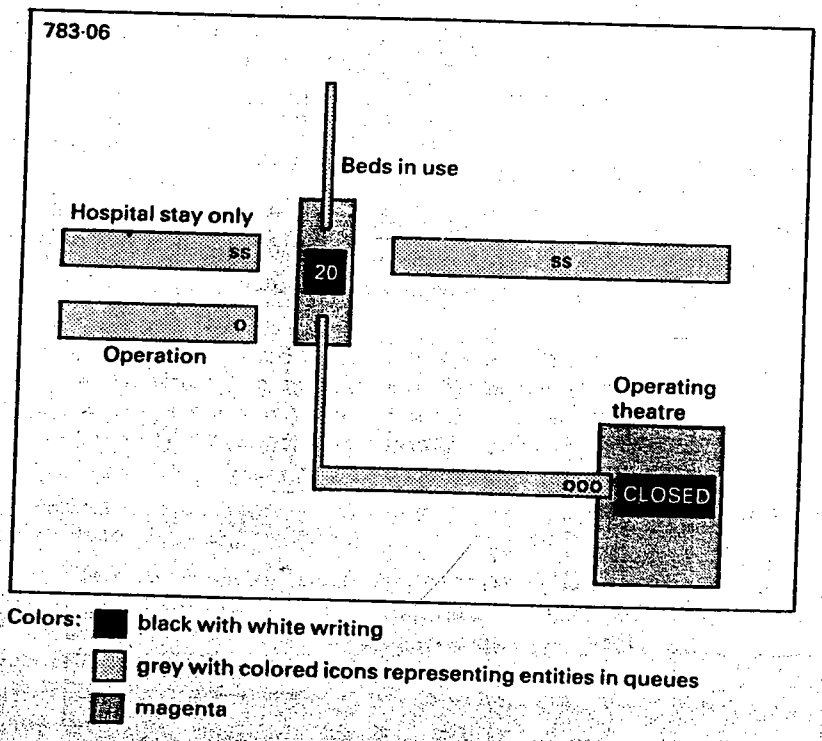

Figure 10.1 Visual display for hospital; time is 178.20 hours after start of simulation with 20 beds in use

it useful to review the specifications for the two examples, as presented in Chapters 1, 2 and 6).

Designing visual displays to get the correct level of detail, good colors and a useful background to the simulation, may demand as much artistic skill as programming skill. The first thing to do in designing a picture is to list what needs to go on the screen. Once this is done, sketch the picture on paper, and when a design is ready (probably following a number of attempted designs and many discarded pieces of paper), produce a version on graph paper. This will provide the co-ordinates necessary for the various visual routines. In the long run, this will be much less time-consuming than repeatedly changing the simulation program.

An alternative is to use one of many ancillary graphics packages (which are becoming increasingly available) to design background displays. A number of packages allow pictures to be drawn, saved, and then retrieved from within another program. Other packages provide for maintenance and manipulation of sprites or icons, which can be used to form the dynamic part of the display. We shall not review this option here, but readers wanting to develop visual simulations professionally, should consider it.

Figures 10.1 and 10.2 respectively, show finished drawings of output from the hospital and repair shop simulations.

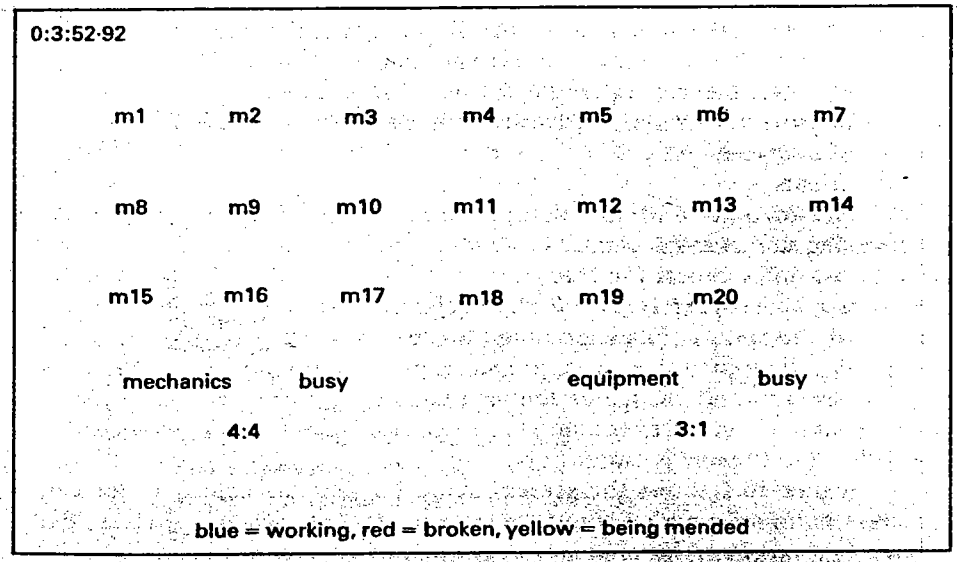

Figure 10.2 Visual display for repair shop; time is 3 hours 52.92 minutes with all four mechanics busy and one of three sets of equipment in use

10.1 PORTRAYING HOSPITAL

10.1.1 Design

Because the hospital simulation is customer oriented, the picture should focus on the path that patients take through the system, and the size of queues for beds and theatre. We need to display:

(*a*) the queue of hospital stay patients for beds (*q*1),
(*b*) the queue of operation patients for beds (*q*2),
(*c*) the queue of operation patients for the operating theatre (*q*3),
(*d*) the number of beds in use,
(*e*) the present state of the operating theatre – open or closed, and if open, whether available or unavailable.

There is little point in displaying *q*4, since it is a dummy queue with no resource constraints.

The picture shows movement of patients going from left to right and top to bottom. Thus *q*1 and *q*2 are placed to the left of the screen, the beds in the middle, and the operating theatre to the right, so that the flow of patient entities which arrive, obtain a bed and have an operation, moves from left to right. Patients returning from the operating theatre to their bed re-enter the bed area from the top of the screen.

In this design, the resource numbers are shown at the right of the queues to which they relate. Thus the display of the number of beds in use is placed just to the right of the arrival queues, *q*1 and *q*2.

We have to decide upon the characters that should represent the two types of patient. Although it is often easiest to use the first letter of the name of each class, here we use *o* for operation patients, and *s* for hospital-stay patients.

The annotation on the picture should be kept to a minimum while conveying sufficient information to understand the simulation display. The resources and queues, for example, need to be labelled. In this display, there is a label saying 'Beds in use', close to the box representing the hospital ward and another label denoting the theatre. Further labels distinguish the two patient-arrival queues. Messages give information about the state of the entities and change as the simulation progresses. For example, a single number indicates the number of beds in use and a word indicates whether the theatre is closed, open, or in use.

Figure 10.1 shows the screen layout but not the colors. Color can convey extra information about the state or class of each entity. Here, the letters representing hospital-stay-only patients remain one color throughout but letters representing the operation patients change color after the operation activity.

The choice of color is a personal matter. The patients are represented by blue which is a neutral color, unlike red, for instance, which is generally associated with stop or danger. The queue backgrounds are white, which is easy to see.

Both patient classes are given the color blue in the class table thus:

```
enter_class (1,'s',blue);
enter_class (2, 'o',blue);
```

This default color can be altered by changing **col** in the entity record. The program distinguishes patients who are ready for an operation by changing their color to **yellow**. This is achieved by placing

```
current^.col := yellow;
```

in the *end_pre_operative_stay* event.

If the background of a picture is black, then yellow is a good contrast for any annotation or legends. This is used here, together with magenta for the blocks of color representing the ward and theatre.

The final touch, which was recommended in the previous chapter, is to place the time clock in the picture. Because the hospital runs continuously, ignoring any breaks for the weekend, holidays, etc., a single point value is used rather than the days: hours: minutes version that can be obtained using **write_time**.

10.1.2 Program details

Within the program, the visual display is updated in two separate places. The appropriate bound or conditional events update the queues and the **display** procedure updates the display of the clock and the number of beds available.

When a queue is updated in a bound event, the **current** entity is moved into a queue using the **move_h** procedure and the new queue is rewritten with **write_queue**. This is best seen in the feeders, *patient1_arrives* and *patient2_arrives*, which both have the format:

```
procedure arrival;
  begin
    give_tail < (current to queue >;
    move_h < current into queue >;
    write_queue < the queue >;
    cause < another arrival >;
  end;
```

Queues are rewritten in the conditional events, following the removal of an entity.

The operating theatre is handled rather differently. The event that

opens the theatre writes 'OPEN' to the screen; conversely, *close_theatre* overwrites it with 'CLOSED'. Similarly, the conditional event *start_operation* writes 'IN USE' to the display. This is erased (by writing out an empty string) when the operation is over.

The number of beds in use is written out within **display** at the end of each time beat. The procedure **display** is called after the scan of the conditional events. The picture is then delayed, so that it may be viewed more easily using a record of the time of the last event, *old_tim*, as explained in Chapter 9.

10.1.3 Steady state versus transient pictures

In Chapter 6, the cumulative moving average method was used to find the start of the steady state for the hospital simulation. The decision about whether to display the simulation only in the steady state, or in the initial period as well, will depend upon the purpose of the display and what information it is being used to convey. If only a steady-state picture is required, then a run-in period should be used during which no display is shown; the picture is initialized after the run-in. However, normally visual output is used to watch how a system behaves over a transient period, as well as the steady state.

10.2 PORTRAYING REPAIR SHOP

10.2.1 Design, with details

Whereas patients in the hospital move through the system, machines in the repair shop remain static; repair men go to the machines. Thus for machine-oriented simulations like this, the approach to a visual display is rather different. Figure 10.2 shows that we can represent each machine as a letter, placing the letters on the screen as if it were a birds-eye view of the factory floor.

To identify each machine in the display, the entities representing each machine need individual co-ordinates. Then, within the simulation, we can access the position of the displayed machine via the entity record. This is best done by extending the entity record with two attributes, as explained in Chapter 7, section 7.3. Here, the repair shop program assumes that the entity record has had

```
x,y :cardinal;
```

added to it, where (x,y) is used as a co-ordinate pair.

Each machine is placed 10 character positions apart from its neighbors in three rows of seven, seven, and six machines, where each row is four positions apart. Within the **initialize** procedure, a simple algorithm is used to achieve this:

```
j := 0;
for i := 0 to < number of machines−1 > do
  begin
    k := i mod 7;
    if k=0 then j := j+4;
    < generate a machine, entity e >;
    e^.x := 10+k*10;
    e^.y :=j;
    < write entity e at x,y >
  end;
```

This algorithm lays out letters on the screen for any number of machines; the horizontal spacing (10) and vertical spacing (4) can be changed easily. The attribute number of each machine is written out beside each letter to help the viewer understand the display.

Each machine is always in one of three states: working, waiting to be mended, and being mended (see the activity diagram in Chapter 2). Different colors are used to show the different states. The default color for *working* is **blue**. This is entered in the class table thus

```
enter_class (1,'m',blue);
```

where *m* is the letter representing the machine. This then changes to **red** when the machine breaks down (an obvious choice of color for conveying that something needs attention), and **yellow** when the machine is actually being mended. For instance, when the machine breaks down, its color is altered by

```
with current^ do
  begin
    col := red; write_entity (x,y,current);
  end;
```

The screen displays the meaning of each color and the total numbers of mechanics and equipment in use (these are written as pairs of numbers in the **display** procedure).

For a more realistic display, repair men and equipment could be shown moving around the machines attending to the broken ones, but this would convey no additional useful information. They would also need to be modelled individually (i.e., represented as entities) rather than as resources. This is a typical example of a situation where an increase in the quality of the display would require items previously modelled as

resources, to be modelled as entities. In many visual simulations all items have to be entities.

Finally, if desired, the effect of the display produced could be enhanced by changing the background to **white** and all foreground annotation to **black**. This can be done by altering the **reset_colors** procedure so that all the visual routines reset the background to **white** and the foreground to **black** i.e.

```
procedure reset_colors;
  begin
    set_foreground (black);
    set_background (white);
  end;
```

10.3 SUMMARY

This chapter shows how to program visual display simulations for the two case studies. The picture for the hospital simulation is based on the activity flow diagram. The activities are portrayed as boxes. Letters representing the patients appear to join queues, enter and leave activities until, when the patient entities are discharged from hospital, they leave the screen. Colors are used to highlight words and to distinguish entities as they move through the system. The visual simulation shows the system in its transient and steady state.

In the repair shop example, the approach is different. The machines are not seen to move through their life cycles but remain static. A change of state is shown simply by a change in color. The entity record is changed to identify the locations of each machine.

These examples are rather inconsequential and could be enhanced by including a display of statistics. Furthermore, the visual impact could be improved by the use of graphics and windowing mechanisms. However, the examples do show how instructions can be incorporated in the simulation logic in order to provide visual output.

10.4 EXERCISES

1. Implement the *hospital system* and *repair shop system* visual simulations, described in this chapter.

   (*a*) Play around with them, changing color and layout, messages and delay factors.

   (*b*) If your computer can write to more than one screen then design some block histograms, showing the current status of the queues, to appear on the second screen.

2. Improve the display of the hospital by using procedures to move entities from right to left and from bottom to top (see Exercise 3, Chapter 9).

3. *Bank System* (Appendix 1.A.1). Using the coding developed in previous exercises, design and implement a visual simulation in Pascal_SIM.

4. *Traffic Light System* (Appendix 1.A.2). Using the coding developed in previous exercises, design and implement a visual simulation in Pascal_SIM.

* * *

10.A HOSPITAL SIMULATION WITH VISUAL OUTPUT

```
program simulate (output);

const
  amount_of_beds       = 20;
  patient1_seed        = 1;
  patient2_seed        = 2;
  hospital_stay_seed   = 3;
  pre_op_stay_seed     = 4;
  operation_seed       = 5;
  post_op_stay_seed    = 6;
  theatre_open_time    = 4;
  theatre_closed_time  = 20;

var
  bed          :bin;
  q1,q2,q3,q4  :queue;
  theatre      :record
                  body              :entity;
                  open,available :boolean;
                  { true if theatre is open, available }
                end;
  old_tim      :real;

{ B events }
procedure patient1_arrives; { stay } { B1 }
  begin
    give_tail (q1,current);
    move_h (12,2,10,current,white);
    write_queue (22,12,white,q1,20);
    cause (1,new_entity(1,1),negexp(6,patient1_seed));
  end;
```

```
  procedure patient2_arrives; { operation } { B2 }
    begin
      give_tail (q2,current);
      move_h (14,2,10,current,white);
      write_queue (22,14,white,q2,20);
      cause (2,new_entity(2,1),negexp(12,patient2_seed));
    end;

  procedure end_hospital_stay; { B3 }
    begin
      return (bed,1);
      move_h (12,40,70,current,white);
      dis_entity (current);
    end;

  procedure end_pre_operative_stay; { B4 }
    begin
      give_tail (q3,current);
      current^.col := yellow;
      move_v (30,14,20,current,white);
      move_h (20,30,50,current,white);
      write_queue (60,20,white,q3,30);
    end;

  procedure end_operation; { B5 }
    begin
      theatre.available := true;
      gotoxy (63,21); write ('      ');
      move_v (30,4,10,current,white);
      give_tail (q4,current);
    end;

  procedure end_post_operative_stay; { B6 }
    begin
      return (bed,1);
      move_h (12,40,70,current,white);
      dis_entity (current);
    end;

  procedure open_theatre; { B7 }
    begin
      theatre.open := true;
      gotoxy (63,20); write ('OPEN  ');
      cause (8,current,theatre_open_time);
    end;

  procedure close_theatre; { B8 }
    begin
      theatre.open := false;
      gotoxy (63,20); write ('CLOSED');
      cause (7,current,theatre_closed_time);
    end;
```



```
{ C events }
procedure start_hospital_stay; { C1 }
  begin
    while (bed.num_avail>0) and (not empty(q1)) do
      begin
        acquire (bed,1);
        cause (3,take_top(q1),negexp(60,hospital_stay_seed));
        write_queue(22,12,white,q1,20);
      end;
  end;

procedure start_pre_operative_stay; { C2 }
  begin
    while (bed.num_avail>0) and (not empty(q2)) do
      begin
        acquire (bed,1);
        cause (4,take_top(q2),negexp(24,pre_op_stay_seed));
        write_queue (22,14,white,q2,20);
      end;
  end;

procedure start_operation; { C3 }
  begin
    with theatre do
      while open and available and (not empty(q3)) do
        begin
          available := false;
          cause (5,take_top(q3),
                 normal(0.75,0.25,operation_seed));
          gotoxy (63,21); write ('IN USE');
          write_queue (60,20,white,q3,30);
        end;
  end;

procedure start_post_operative_stay; { C4 }
  begin
    while (not empty(q4)) do
      begin
        cause (6,take_top(q4),negexp(72,post_op_stay_seed));
      end;
  end;

procedure display;
  var
    i :cardinal;
  begin
    gotoxy (30,12); write (bed.number-bed.num_avail:2);
    gotoxy (1,1); writeln (tim :7:2);
    gotoxy (1,1);
    for i := 1 to trunc ((tim-old_tim)/2) do delay;
    old_tim := tim;
  end { display };
```

```
procedure run(duration:real;max_C:cardinal);
  var
    c :cardinal;
  begin
    running := true;
    repeat
      if calendar=calendar^.next then running := false
        else
          begin
            display;
            tim := calendar^.next^.item^.time;
            if duration<tim then running := false
              else
                begin
                  while (calendar<>calendar^.next) and
                        (tim=calendar^.next^.item^.time)  do
                    begin
                      calendar_top;
                      case current^.next_B of
                        0: ;
                        1: patient1_arrives;
                        2: patient2_arrives;
                        3: end_hospital_stay;
                        4: end_pre_operative_stay;
                        5: end_operation;
                        6: end_post_operative_stay;
                        7: open_theatre;
                        8: close_theatre;
                      end;
                    end;
                  for c := 1 to max_C do
                    case c of
                      1: start_hospital_stay;
                      2: start_pre_operative_stay;
                      3: start_operation;
                      4: start_post_operative_stay;
                    end;
                  end;
          end
    until not running;
  end { run };

procedure initialize;
  begin
    make_sim; make_streams;
    make_bin (bed,amount_of_beds);
    make_queue (q1); make_queue (q2);
    make_queue (q3); make_queue (q4);
    with theatre do
      begin
        body := new_entity(3,1);
        open := true; available := true;
        cause (8,body,theatre_closed_time);
      end;
  end { initialize };
```



```
procedure picture;
  begin
    make_screen;
    set_background (black);
    enter_class (1,'s',blue);
    enter_class (2,'o',blue);
    clear_screen;
    write_block (28,10,32,14,magenta);
    write_block (60,18,70,23,magenta);
    set_foreground (yellow);
    gotoxy (4,11); write ('Hospital stay only');
    gotoxy (4,15); write ('Operation');
    gotoxy (32,8); write ('Beds in use');
    gotoxy (60,15); write ('Operating');
    gotoxy (60,16); write ('Theatre');
    reset_colors;
  end { picture };

procedure report;
  begin
  end { report };

begin
  initialize;
  picture;
  cause (1,new_entity(1,1),0);
  cause (2,new_entity(2,1),0);
  old_tim := 0;
  run (365,4);
  report;
  reset_colors;
end.
```

10.B REPAIR SHOP SIMULATION WITH VISUAL OUTPUT

To implement this, *x* and *y* must be included in the entity record and initialized in new-entity.

```
program simulate (output);

const
  amount_of_machines    = 20;
  amount_of_mechanics   = 4;
  amount_of_equipments  = 3;
  inter_break_down_time = 60;
  breakdown_seed        = 11;
  removal_time_seed     = 12;
  repair_time_seed      = 13;
```

```
  var
    at_work              :boolean;
    q1,q2,q3             :queue;
    mechanic,equipment   :bin;
    old_tim              :real;


  { B events }
  procedure end_working; { breakdown } { B1 }
    begin
      if at_work then { machine can't break down after shift }
        begin
          with current^ do
            begin
              col := red; write_entity (x,y,current);
            end;
          give_tail (q1,current);
        end;
    end;

  procedure end_removal; { B2 }
    begin
      give_tail (q2,current);
    end;

  procedure end_repair; { B3 }
    begin
      return (mechanic,1);
      return (equipment,1);
      with current^ do
        begin
          col := blue; write_entity (x,y,current);
        end;
      if at_work then
        cause (1,current,
               negexp(inter_break_down_time,breakdown_seed))
      else
        begin
          gotoxy (40,1);
          write ('Finished repair at ');
          write_time;
        end;
    end;

  { C events }
  procedure start_removal; { C1 }
    begin
      while (mechanic.num_avail>0) and (not empty(q1)) do
        begin
          acquire (mechanic,1);
          cause (2,take_top(q1),normal(5,1,removal_time_seed));
        end;
    end;
```



```
  procedure start_repair; { C2 }
    var
      e :entity;
    begin
      while (equipment.num_avail>0)
        and (not empty(q2)) do
        begin
          e := take_top (q2);
          with e^ do
            begin
              col := yellow; write_entity (x,y,e);
            end;
          acquire (equipment,1);
          cause (3,e,normal(8,2,repair_time_seed));
        end;
    end;

  procedure display;
    var
      i :cardinal;
    begin
      reset_colors;
      gotoxy (20,20);
      with mechanic do
        write (number :1,':',number-num_avail :1);
      gotoxy (60,20);
      with equipment do
        write (number :1,':',number-num_avail :1);
      for i := 1 to trunc ((tim-old_tim)/2) do delay;
      old_tim := tim;
      gotoxy (1,1); write_time;
    end;

  procedure run (duration :real; max_C :integer);
    var
      c :integer;
    begin
      running := true;
      repeat
        if calendar=calendar^.next then running := false
        else
          begin
            tim := calendar^.next^.item^.time;
            display;
            if duration<tim then running := false
            else
              begin
                while (calendar<>calendar^.next) and
                      (tim=calendar^.next^.item^.time)  do
                  begin
                    calendar_top;
```

```
                    case current^.next_B of
                      0: ;
                      1: end_working;
                      2: end_removal;
                      3: end_repair;
                    end;
                  end;
                for c := 1 to max_C do
                  case c of
                    1: start_removal;
                    2: start_repair;
                  end;
              end;
          end
    until not running;
  end { run };

procedure initialize;
  var
    i,j,k :cardinal;
    e     :entity;
  begin
    make_sim;
    make_streams;
    make_bin (mechanic,amount_of_mechanics);
    make_bin (equipment,amount_of_equipments);
    make_queue (q1); make_queue (q2); make_queue (q3);
    { set a shift working }
    at_work := true;
    { set breakdown times for all machines }
    j := 0;
    for i := 0 to amount_of_machines-1 do
      begin
        k := i mod 7;
        if k=0 then j := j+4;
        e := new_entity(1,i+1);
        cause (1,e,
           negexp(inter_break_down_time,removal_time_seed));
        with e^ do
          begin
          x := 10+k*10; y := j;
          write_entity (x,y,e);
          write (i+1 :1);
          end;
      end;
  end { initialize };
```



```
procedure picture;
  begin
    make_screen;
    enter_class (1,'m',blue);
    reset_colors;
    clear_screen;
    gotoxy (10,17); write('mechanics      busy');
    gotoxy (50,17); write('equipment      busy');
    gotoxy (12,24);
    write('blue=working, red=broken, yellow=being mended');
  end { picture };

begin
  picture;
  initialize;
  old_tim := tim;
  { run for 8 hours, in minutes }
  run(8*60,2);
  { close down the shift }
  at_work := false;
  gotoxy (20,1); write ('end of shift');
  { run the simulation out, ie. do all repairs left
    10000 is arbitary large number }
  run (10000,2);
  reset_colors;
end.
```

11

Modelling Complexities

Simulations are made up of building blocks of simple queueing systems which can be linked in parallel or in series, or both, to describe complex systems. However, the provision of links between simple queueing systems is not always adequate. Complexity in modelling terms arises when this structure has to be adapted in various ways. For example, when:

(*a*) arrival rates or resource provisions are dependent on the time of day,
(*b*) feedback from one part of a model influences activities in another,
(*c*) entities leave queues without taking part in the activity for which they have been queueing,
(*d*) entities have to be interrupted in the middle of one activity in order to start another,
(*e*) an entity can take part in more than one activity at the same time.

These and other complexities may need to be introduced in order to increase the validity of a simulation. In this chapter we present some of the methods for dealing with these sorts of complexities.

11.1 DEPENDENCE ON TIME

The models built up in the previous chapters assume constant average arrival rates throughout the duration of the simulation. However, most systems describing human behavior vary in a fairly predictable way with time. Shops, banks, buses, traffic systems, and indeed hospitals and manufacturing processes, all have busy and slack times of the day and of the week.

These cycles, if not detected, can result in fallacious simulation models. Information about the arrival rates may either relate to a busy or a slack period, or they may be averaged indiscriminately over both. Simulations which are based on the former are obviously biased, and those based on the later will underestimate queue lengths in the busy periods and overestimate resource requirements at other times.

If the client is concerned about the behavior of the system when it is particularly busy, then a single average arrival rate (relating to the busy time of day) may suffice. However, the simulation will only be helpful if the 'real life' system has time to reach and maintain a steady state during this busy period.

Thus frequently, in order to study the detailed changes in queue length and resource use over time, it is necessary to vary the mean of the arrival distribution with time. The obvious way to do this is to give the sampling function a time-dependent parameter mean. Unfortunately, this gives rise to bias because the change in the average arrival rate does not take effect until *after* the next scheduled arrival has occurred. This delay is likely to be long when there is a change from low frequency to high frequency arrivals.

The simplest solution to this problem is called *thinning* (Lewis and Schedler 1979). The change in the average arrival rate over time is assumed to be a piecewise continuous function such as that shown in Fig. 11.1. Let the function be $f(t)$ with a maximum value, m. In the method of thinning,

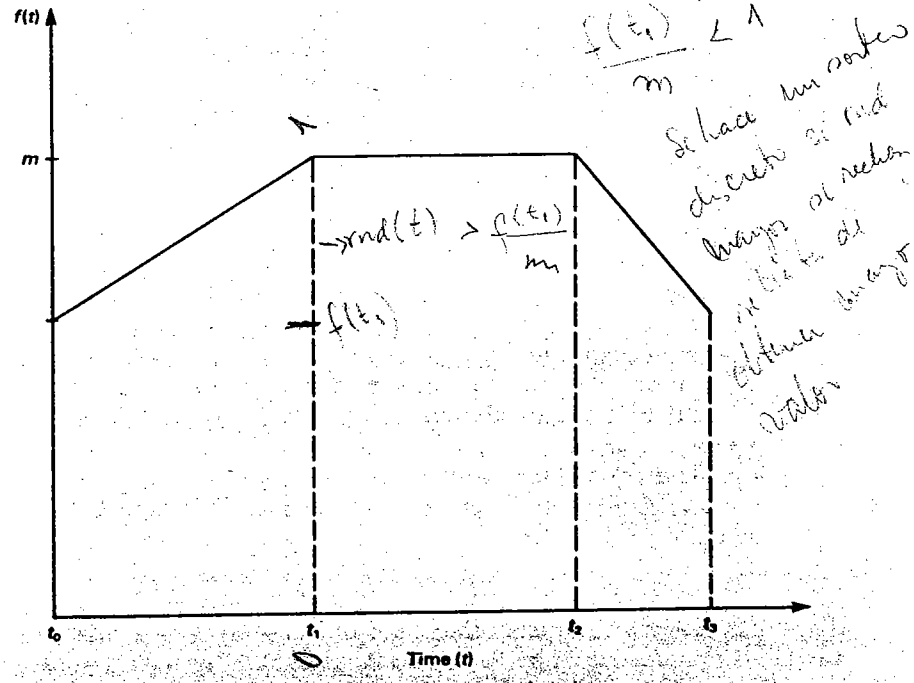

Figure 11.1 Example of typical piecewise continuous function, representing changing arrival rate over time where maximum value of f(t) is m

arrivals are generated at the maximum rate m and then some of these are *pruned*. The probability of an arrival being pruned depends on the value of $f(t)$ at the time of the arrival.

The arrival rate at any point in time is assumed to be Poisson. Sampling is a two-stage iterative process as follows.

(*a*) A sample is generated from the negative exponential distribution with mean $1/m$ (i.e., at maximum rate m). This determines the time, u, between two arrivals. If t is the time that the last arrival occurred (or was due to occur at this maximum arrival rate), then $t + u$ is the time for the next arrival.

(*b*) To decide whether this sampled arrival should take place or be *pruned out*, a random number between 0 and 1 is sampled and if it is greater than $m/f(t + u)$, then the arrival is rejected, otherwise it is accepted. Thus the thinning takes place at the time the arrival is *due* and not at the current simulation time. If the arrival is rejected, the time $t + u$ is set to t and the two stages are repeated until an arrival is generated.

It is easy to check that, using this method, the expected arrival rate at any point in time is $f(t)$. Thinning is an iterative method which slows down the sampling in proportion to the difference between m and $f(t)$. A function to sample the time of the next event is coded as follows:

```
function thintime: real;
var
  t, u: real;
begin
  t := tim;
  repeat
    u := negexp(1/m,s);
    { where s is the sampling stream and
      and m is the maximum arrival rate }
    t := t + u;
  until rnd (s) < (fun(t)/m);
  { where fun is the function of arrival rate with time }
  thintime := t - tim;
end;
```

Algorithms using techniques other than thinning are also available. One example is that described by Klein and Roberts (1984) which is used in the simulation package INSIGHT (Roberts 1983). However, the advantage of thinning is that it is both efficient and easy to understand.

11.2 FEEDBACK

Feedback occurs when the activities in one part of a model are dependent on entity numbers or activities in another part of a model. For example, in queueing systems, servers may be withdrawn from a system when queues are short (or vice versa) or in population models, the number of births will invariably be dependent on the number of adult females in the system.

Although there are no standard techniques for describing feedback in activity-flow diagrams, the logic is fairly straightforward; the variable factor subject to feedback is made dependent on a count of the appropriate group of entities.

Feedback often has the effect of stabilizing systems: keeping queue lengths from becoming excessively long or preventing the severe underutilization of resources. It will thus reduce the variance of key measurements. However, the additional variables and their complex interactions make it much more difficult to perform sensitivity analyses and to interpret the results from simulation runs.

11.3 QUEUE BEHAVIOR

In many simulations, once entities join a queue they remain there until they are served. However, if the simulation is to describe the activities of people, it may have to model queueing behavior which is much less predictable. People fail to join queues, leave them to join other queues which they perceive to be shorter and, quite unpredictably, leave queues. If these activities are found to be relatively infrequent, or those running the simulation are much more interested in resource use than queueing problems, then it may be possible to ignore them without significantly affecting the results of the simulation. However, if the simulation has to provide detailed information about queue lengths and waiting times, some or all of these different queueing activities may have to be modelled.

11.3.1 Balking

Failure to join queues, called *balking*, is very common. People generally exhibit this behavior when a queue, or the perceived waiting time for service, is long. It is, therefore, an example of feedback in the system which tends to stabilize it and to reduce the overall variance.

Balking can easily be incorporated in a simulation model by making the addition of entities to queues conditional on queue length. Difficulties

arise in deciding what queue length to choose, because it will vary from individual to individual and from day to day. Additional data may therefore be needed to provide a distribution from which to sample. Balking is a more complicated phenomenon than appears at first sight because those who balk the queue on one occasion may well join it on a future occasion when the queue is shorter. Those collecting data are likely to count these people twice, thus inflating the number of potential arrivals in the system. This will certainly be misleading if the simulation is used to predict the effect of changing certain factors, such as the number of servers in the system. This may be counterbalanced to some extent by undetected balking by those passing by unnoticed or hearing from others about the long queues.

11.3.2 Queue swopping

Queue swopping occurs where people have a choice of queues. Assuming people join the shortest queue, they may swop queues if another queue moves faster and becomes shorter than the one they are in. One occasion when it invariably occurs is when one server becomes free while there are still queues for the other servers. If queue swopping is ignored, server-busy time will be underestimated and queueing time overestimated. It is particularly important to take account of this phenomenon when assessing the benefits of multiple queues as opposed to single queues in, for example, banks and similar institutions.

In the three-phase approach, queue swopping can be described in an extra conditional event, which might be called *queue_swop*. This would swop people between queues according to certain rules: for example, if a server were free, or if the difference between the lengths of any two queues were two or more people. This conditional event would be the last to be tested in the executive in order to ensure that it took place at the end of each time beat.

This description of queue swopping is clearly a simplification of what happens in practice because it assumes that people behave logically, that they can see the length of the other queues, and that they base their actions on queue length rather than on the perceived length of time they will take to get served. If these assumptions are inappropriate, then the rules in the conditional event, *queue_swop*, can be changed accordingly.

Using event scheduling, a procedure called *queue_swop* can be programmed, and called from within any scheduled event whose execution may result in state changes which give rise to queue swopping.

11.3.3 Reneging

The probability of a person leaving a queue, or *reneging*, before reaching the server, will depend on the time they have spent in the queue, whether they are aware of their place in the queue (which is not true, for example, of queueing telephone calls) and, if so, where they are in the queue.

The probability of reneging must thus be related to the individual entities. To describe this in a simulation, the program is given a bound event, which might be called *renege*, whose function is to withdraw the entity from the queue. On joining the queue, the entity is also put in the calendar to set the time for reneging. If the bound event, *renege* takes place before the entity reaches the server, then it is removed from the queue. If, on the other hand, the entity reaches the server before *renege* takes place, then the entity is removed from the calendar instead. The removal of an entity from the calendar is called 'fetching'.

11.4 FETCHING

In some simulation models, activities must be prematurely terminated or interrupted, such as reneging in the previous section. The entities are *fetched*, or *descheduled*, from the calendar. In the machine repair problem, for example, the mechanics may stop what they are doing for lunch. A bound event would be set to identify lunch time, but at that point in the simulation, there might be *repair* activities already in progress. A procedure to fetch or deschedule an entity would have the following pseudocode:

```
procedure fetch (e :entity);
  begin
    if < the first entity in the calendar does not equal e > then
    repeat
      < look at the next entity in the calendar >
    until < the entity equals e >;
    < take the entity out of the calendar >;
  end;
```

When an entity has been fetched, it must be put in a queue to await the restarting of the activity or the start of a new activity and the resources must also be released. In the example described above, the resources would be committed to a new activity: *eating_lunch*.

If the activity is to be continued after an interruption, then the simulation must remember the time remaining to be spent in that activity. This can be held as an entity attribute and recalled when the activity is restarted.

When modelling reneging and fetching, the simulation may be slowed down considerably by the need to search the queue or calendar to extract the entities, especially if the list is long. This problem can be overcome by

providing the entity record with information about the location of the entity in the queue, but this increases the complexity of the entity record. In Pascal_SIM, the entity can be made to point at the links in the queue, in addition to the links pointing at entities. If the entity is only ever likely to be in *one* queue, then the structure of the entity record is as follows:

```
type
  an_entity = packed record
                avail        :boolean;
                attr, next_B :cardinal;
                class        :class_num;
                time         :real;
                q_place      :link;
              end;
```

The function *q_place* is set to point to *nil* in **make_entity**. In **give**, coding must be included to set *q_place* to the link and in **take** it must be made to point to *nil* again. Suppose we want to take the current entity from queue, then the function **take** is called as follows:

```
var
  e :entity;
  e :take(q,current^.q place)
```

where *e* is an entity. The problem becomes much more complex if an entity is allowed to be on more than one queue at one time. The entity record must allow enough pointers to links for *all* the queues that the entity may be on. This can be done by providing an array of links, one for each queue in the simulation.

11.5 SHADOW ENTITIES

Sometimes entities must take part in two or more activities simultaneously. For example, the theatre in the hospital simulation takes part in the *operation* activity, but is also scheduled to *open* or *close*. Thus the theatre can be regarded as simultaneously taking part in two sets of activities:

(*a*) operation
(*b*) open or closed.

Similarly, if the repair shop simulation was extended to model the actual processing operation of the machines, then the machines need to be modelled by two sets of activities:

(*a*) processing
(*b*) working and broken down.

In the hospital example, if the theatre needs to be closed when an operation is in progress, then the operation is allowed to continue until its scheduled time of completion. As was seen in Chapter 6, this can be quite easily modelled with a simple Boolean variable. However, this is not possible in an extension to the repair shop example; when a machine breaks down while it is processing, the processing activity must be halted immediately.

Two sets of concurrent activities like this can be modelled by using two separate but related entity types. These are called the *parent* and the *shadow* entities. The parent entities engage in the more frequent routine activities and each may have more than one shadow. The shadow entities engage in the concurrent activities.

The shadow entities must have the facility to communicate with their parent entities so that when something happens to the parent entity that affects the shadow, or *vice versa*, the related entities can be found quickly and descheduled or removed from queues. For instance, if a machine has a parent entity engaged in an activity and a shadow entity scheduled for a breakdown time, then when a breakdown occurs the parent entity must be fetched from the calendar.

To enable a shadow entity to find its parent entity, the shadow entity record is made to point to its parent. The parent entity similarly points to its shadow. In a complex simulation, a parent entity may be on more than one list and have several shadows waiting to trigger events. The activities of one shadow entity may thus affect the other shadows as well as the parent. The parent entity will thus need to point to all the shadows. Furthermore, when the entities are found they may need to be removed from activities or queues. Davies and Davies (1987) describe how they used pointers in Pascal_SIM for their simulation of patients with end-stage renal failure.

11.6 CELLULAR SIMULATION

In the three-phase approach, every conditional event is scanned at least once in every time beat. This is inefficient because only a few will need to be activated, and sometimes none. On some occasions a C event activates a B event in the same time beat, causing all the C events to be tried and retried in one time beat. This redundancy is not present in the event scheduling approach where all conditional events are included in the scheduled events.

The method of cellular simulation, devised by Spinelli de Carvalho (1976), overcomes these inefficiencies. In this method, each C event is deemed to be either *open* or *closed*. When the executive scans the events, it will only test the conditions of those that are open. Each B event must

open all the C events that could possibly be affected by its execution and close all others. This technique maintains the modularity of the three-phase method while considerably increasing its efficiency by identifying the relationship between the B and C events.

Cellular simulation can be implemented with Pascal_SIM by declaring a Pascal set that can include all the conditional event numbers. For example, in the hospital simulation, which was four conditional events, we can declare:

```
type
  C_event =1..4;
var
  open: set of C_event;
```

Within the executive, *open* is initialized as empty after the A phase (the advance of the clock to the next time beat) thus:

```
open := [ ];
```

and in the C phase of the executive, only conditional events that have been opened are called:

```
for c := 1 to max_C do
  if c is open then
    case c of
      1: < etc. >
    end;
```

Within bound events, conditional event numbers are added to the *open* set.

Table 11.1 Cellular simulation applied to hospital example showing conditional events which must be opened following completion of a bound event

Bound event	*Conditional event*	*Numbers*
patient1_arrives	*start_hospital_stay*	[1]
patient2_arrives	*start_pre_operative_stay*	[2]
end_hospital_stay	*start_hospital_stay*	[1,2]
	start_pre_operative_stay	
end_pre_operative_stay	*start_operation*	[3]
end_operation	*start_operation*	[3,4]
	start_post_operative_stay	
end_post_operative_stay	*start_hospital_stay*	[1,2]
	start_pre_operative stay	
open_theatre	*start_operation*	[3]
close_theatre	*none*	[ ]

For example, in the hospital simulation, in the B event *patient1_arrives*, the C event *start_hospital_stay* is opened thus:

```
open := open + [1];
```

Table 11.1 shows the C events that need to be opened by each bound event. This table indicates that, if most B events occur at different points in time, then the saving in scanning time is likely to be more than 50 %. The event *close_theatre*, for example, requires no C event scanning at all. However, if several different B events are likely to occur together on the same time beat, the savings are likely to be reduced considerably.

11.7 SUMMARY

Discrete-event simulations can describe many different systems and are extremely flexible. From all points of view, including data collection, variance reduction, and sensitivity analysis, it is desirable to keep simulation structures as simple as possible and to keep the number of factors small. The first attempts at describing a system should certainly be as simple as possible. However, some systems need the more complex techniques that have been described in this chapter.

Simulation models often require a *time-varying* arrival distribution, since arrival rates can vary considerably across a day, week, or other time period. Particularly in systems which involve humans queueing, it is necessary to model different types of queue behavior including *balking*, *swopping*, and *reneging*. In complex models, entities can take part in more than one activity concurrently. It becomes necessary to use *shadow* entities, and provide facilities for shadow and parent entities to fetch each other from the calendar.

11.8 EXERCISES

1. The number of patients admitted to hospital varies by the time of day: 70 % are admitted in the morning (9 a.m. to 12 p.m.), 25 % in the afternoon (12 p.m. to 5 p.m.), and 5 % in the rest of the 24 hours. Adapt the simple hospital simulation program (using the version developed in Exercise 3, Chapter 5) to take account of this, using the method of thinning.
2. A baker used to make 100 white loaves a day but decides to adjust the supply of large white loaves according to the demand on the previous day. If he has *n* large white loaves remaining at the end of the day, then he makes n/2 fewer (rounded up to the nearest integer) to sell the next day. If he has met demand, then he makes an extra 10 to sell the next day. If demand has a

Poisson distribution with an average of 98, compare the expected sales and number of wasted loaves with those of his previous policy.

3. A hospital waiting list maintains an average of 100 patients. Of these, 5 % move house or die each year. Describe the reneging in pseudo-code under the different assumptions:

   (*a*) the hospital is informed immediately a patient leaves the waiting list,
   (*b*) the hospital is not informed until a patient fails to turn up on their admission day.

   Implement your code in Pascal_SIM in the simple hospital system (using the version developed in Exercise 3, Chapter 5), choosing appropriate parameters for the admission rate and length of stay. State any assumptions that you have to make.

4. A man who is an accurate shot, shoots birds in flight and sends his dog to fetch them. The dog is certain to find the bird if he is with the man when it is shot but has only a 40 % chance of finding it if he is retrieving another bird at that time. Should the man shoot every bird if he wishes to maximize his catch of birds? State the assumptions you are making in answering this question.

5. Six washing machines in a launderette wash clothes for the customers who arrive randomly. These machines break down every now and then and have to wait for repair. They usually break down while loaded with washing, whereupon the washing has to be removed and put in another machine. Use shadow entities to describe the activities in this system.

6. *Bank System* (Appendix 1.A.1). Use the code developed in Exercise 4, Chapter 6, to carry out the following.

   (*a*) Implement the thinning method to describe the variable arrival rate throughout the day.
   (*b*) Improve the model still further by modelling queue swopping and see whether this affects the results and conclusions of the experimentation performed in the exercises at the end of Chapters 6 and 8.
   (*c*) Explain why is it more important to describe the queue swopping rather than reneging or balking in this system.

7. *Traffic Light System* (Appendix 1.A.2). Explain what feedback mechanisms exist in this system and why it is difficult to model them. Discuss what relevance they might have to the objectives of the study.

8. Implement cellular simulation in the three-phase version of the repair shop simulation.

12
The Process View

In Chapter 2 we introduced the reader to basic discrete-event simulation methodology and two world views: event scheduling and the three-phase approach.

In this chapter, an altogether different world view, the *process view*, is presented and explained. The process view can be split into two types: *process interaction*, and a more restricted approach which we shall call *process description* (sometimes called *transaction-flow*, and often wrongly called process interaction). Each of these is considered here.

The process description world view has descended from the Simulation Programming Language (SPL) GPSS, which was originally developed at IBM in the late 1950s. It is discussed in section 12.2.2, where it is used to code a version of the hospital simulation. Process interaction started with the SPL, SIMULA, developed at the Norwegian Computing Centre in the mid 1960s. The facilities that distinguish process interaction from process description are discussed in the following section.

Simple process description models can be built using Pascal. The hospital simulation is recoded using the process view, drawing on some process facilities provided in Pascal_SIM. This is followed by a comparison between the three world views and comments on the benefits of each.

12.1 BASIC CONCEPTS

In the process view, the actions taken by each class of entity are mapped out as a *process*. The process is a description of the entity cycle or flow.

In the hospital case study, for example, there are three classes of entity: hospital-stay-only patients (*patient*1), operation patients (*patient*2), and the *theatre*. Beds can still be modelled as resources. The processes in pseudo-code are:

```
procedure patient1;
  begin
    < cause the arrival of a new patient1 >;
```

```
        < acquire a bed >;
        < stay >;
        < return the bed >
        < leave >;
      end;

    procedure patient2;
      begin
        < cause the arrival of a new patient2 >;
        < acquire a bed >;
        < pre-operative stay >;
        < operation >;
        < post-operative stay >;
        < return the bed >
        < leave >;
      end;

    procedure theatre;
      begin
        repeat
          < open theatre >;
          < wait >
          < close theatre >;
          < wait >
        until < end of simulation >;
      end;
```

This appears very simple compared to the three-phase and event scheduling approaches. Although there is no mention of queues and no explicit queue manipulation, this pseudo-code can be converted to a simulation model using a process-oriented simulation language with little expansion.

In this example, at any one time, there will be multiple instances of entities of type *patient*1 and *patient*2 at some stage in their process description, whereas a single instance of the *theatre* will loop through the states of open and closed until the simulation is halted. Each entity flows through its process until it either leaves the simulation, or the simulation ends. At each stage of its process, an entity is in one of two states:

(*a*) *delayed*, i.e., taking part in an activity such as *stay* or *operation*, or

(*b*) *blocked*, i.e., waiting to enter an activity, acquire some resources, or receive a message from another entity.

There must be interaction between processes, which can be modelled as entities sending messages to each other. For instance, when the *theatre* opens, it will post a message to any entities waiting to enter the activity *operation*.

12.2 PROCESS DESCRIPTION

Most process-oriented languages and packages do not allow direct communication between processes, since this demands facilities that are not available in the common procedural languages on which they are based. Interaction must be described through the use of shared data, normally global variables.

As an example, a global Boolean variable, called *theatre*, could have the value *true* when the theatre is open, and *false* otherwise. Its value would be changed in the *theatre* process, and read within the process *patient*2 to determine whether the theatre was open or not. This compromise approach to process interaction is *process description*, since processes can be described, but entities cannot explicitly interact with entities having different processes.

12.2.1 Time flow mechanism

The simulation executive for process description or process interaction is more complex than the mechanism for either event scheduling or the three-phase approach, because it controls the allocation of resources to entities. The burden of describing queue changes and resource allocation is thus moved from the programmer to the executive.

The process description executive works by maintaining two ordered lists, the calendar of future events (like the calendar for the two- and three-phase executive), and the chain of suspended entities. The latter is a chain of entities which are blocked at some stage in their process description and are waiting in queues. The chain is ordered by their time of arrival in the queue or by priority (see the end of this section). The executive is then very similar to the three-phase approach:

A: < advance time >;
B: < execute bound events off the calendar >;
C: < scan through all suspended entities >;

At each time beat, any entity that comes off the calendar in the B phase moves sequentially through the actions specified in its process description until either:

(*a*) it leaves the simulation;
(*b*) it is delayed, and thus must be entered into the calendar again; or
(*c*) it cannot continue for any reason (for instance, a resource is not available), in which case it is put on the end of the suspended chain.

In the C phase, the scan moves through the chain of suspended entities and tries to execute the next part of their process description, such as engaging them to an activity. Movement of each entity on the suspended chain is tried in turn and when successful, it continues until one of the three conditions above occurs.

A number of different algorithms for a simulation executive can be developed from this and one version is:

```
A: < advance time to the time of the event
   at the top of the calendar >;
B: while < an entity at the top of the calendar has this time > do
     begin
       < remove it >;
       repeat
         < execute its next action >;
       until < it terminates, is entered back in the calendar,
               or is put onto the end of the suspended chain >;
     end;
C: repeat
     < start at top of suspended chain >
       while < not at end of suspended chain > do
         if < entity can execute its next action > then
           begin
             < remove it >;
             repeat
               < execute its next action >;
               until < it terminates, is entered back in the
                       calendar, or is put onto the end of the
                       suspended chain >;
           end;
   until < suspended chain not changed >;
```

It is usual for the executive to enter entities at the tail of the suspended chain and to attempt to reactivate them from the top of the chain, thus maintaining a simple FIFO policy. In more complex cases where queue-priority rules are to be used, then a single measure of priority (as explained in Chapter 7) must be associated with entities from a particular process, so that they can be added to the suspended chain in priority order. Thus, for example, operation patients can be given priority for beds over hospital-stay patients by assigning them a higher number.

Although programming is made easier by using a single suspended chain rather than separate lists for each queue, it has severe disadvantages. The suspended chain can become large, causing the executive to be highly inefficient. In addition, it makes statistics collection and the provision of

visual output much more difficult to implement. These difficulties are discussed in section 12.5.

12.2.2 GPSS

GPSS (the General Purpose Simulation Language) is probably the most widely used simulation programming language in the world. Since its conception and initial development, it has undergone numerous revisions, and there are versions available for many different machines. This section uses the hospital example to show how easy it is to convert the process-oriented pseudo-code given in section 12.1 into a simulation using a process-oriented language.

Figure 12.1 shows a GPSS program for the hospital example. The appearance of GPSS is somewhat archaic, reflecting the time of its inven-

```
          SIMULATE

          STORAGE    BED,20        ;there are 20 beds

*         MODEL SEGMENT FOR HOSPITAL STAY ONLY PATIENTS
          GENERATE   45,15         ;create patients
          ENTER      BED,1         ;acquire a bed
          ADVANCE    30,10         ;stay
          LEAVE      BED,1         ;return the bed
          TERMINATE                ;leave

*         MODEL SEGMENT FOR OPERATION PATIENTS
          GENERATE   16,4          ;create patients
          ENTER      BED,1         ;acquire a bed
          ADVANCE    10,5          ;pre-operative stay
          SEIZE      THEATRE       ;queue for available theatre
          GATE LS    OPEN          ;use it if open
          ADVANCE    1             ;operation
          RELEASE    THEATRE       ;return the theatre
          ADVANCE    7.5,2.5       ;post-operative stay
          TERMINATE                ;leave

*         MODEL SEGMENT FOR THEATRE
          GENERATE   ,,1           ;create 1 theatre
AGAIN     LOGIC S    OPEN          ;set logic gate
          ADVANCE    8             ;theatre open
          LOGIC R    OPEN          ;reset logic gate
          ADVANCE    40            ;theatre closed
          TRANSFER   AGAIN         ;goto again

*         TIMER SEGMENT
          GENERATE   336           ;generate a dummy transaction
          TERMINATE  1             ;decrement run control counter

          START      1             ;start, run control counter=1
          END
```

Figure 12.1 Hospital example in GPSS; lines that begin asterisks, and text preceded by semi-colon, are comments

tion. All statements must be in upper-case and the column positions of each statement and its parameters are vital in most implementations.

In GPSS, a statement that can act upon entities (i.e., is part of a process description) is called a *block*, and each process is called a *segment*. Entities that *flow* through a segment are called *transactions*. In Fig. 12.1 we have the following.

(*a*) *Generate* specifies the arrival distribution for the process, with the first two parameters representing the mean and spread of a uniform distribution.

(*b*) *Enter* and *seize* are equivalent to **acquire** in Pascal_SIM, where *seize* is applied to a single-server resource.

(*c*) *Advance* results in a time delay for the transaction.

(*d*) *Leave* and *release* are equivalent to **return** in Pascal_SIM, where *release* is applied to a single server that has been *seized*.

(*e*) *Terminate* has the same effect on a transaction as the Pascal_SIM routine **dis_entity** has on an entity, i.e., it is removed from the simulation and destroyed.

Resources have to be declared if more than one resource is to be made available, using the statement *storage*. If only one resource is used, as with the *theatre*, no declaration is necessary.

In the GPSS example, a *logic gate* (a GPSS construct similar to a global Boolean variable in Pascal) called *open*, is used to control the opening and closing of the theatre. The *logic s* block in the theatre segment puts *open* to *set*. The *gate ls* block in the segment for the operation patients, requires than *open* is *set* for the transaction to continue. If this is true, the transaction can proceed and try to *seize* the theatre. The theatre is closed by the *logic r* block, which puts *open* to *reset*. Now transactions in the *operation patients* segment will be blocked by *gate ls*. Note how the *theatre* is modelled by a *transfer* to a label called *again*. (GPSS does not provide any structured constructs, such as a *repeat...until* loop.)

A GPSS program usually has a special segment called a *timer segment* which controls the length of simulation run, statistics collection, and other details. Figure 12.1 includes a timer segment and run control details for the hospital example. A dummy transaction arrives after 336 hours (14 days of simulated time), and its termination by *terminate* 1 decrements a run control counter established with the *start* statement. When decremented to zero, the simulation halts. Readers interested in learning more about GPSS should consult Schriber (1974).

12.3 PROCESS INTERACTION

A true process interaction language, such as SIMULA (Birtwistle *et al.* 1979), provides a number of facilities not normally found in common high-

level languages, such as Pascal, or process description languages such as GPSS. In particular they provide the following.

(*a*) *Communication facilities*, so that entities can influence entities from other processes without resorting to the use of intermediate data structures. In SIMULA, a process can directly *passivate* (i.e., suspend) or *activate* (i.e., set into action again) another process. In GPSS, transactions can only influence each other through intermediate data structures such as the logic gate.

(*b*) A *co-routine facility* which allows entity attributes to be declared locally to a process and retained between activations of the process. This contrasts markedly with GPSS, where attributes are associated with a transaction rather than a segment. This is much less flexible, because every transaction generated in a segment has to have the same attributes and there is no way of telling what attributes are being used by transactions from different segments.

(*c*) *Inheritance* so that a process can inherit the attributes and the local routines of other processes. For instance, particular types of patient (e.g., operation, hospital stay, emergency) can inherit from a patient class all the attributes common to all patients (e.g., name, weight, age, etc.).

Any language which provides all of these is now referred to as *object-oriented*, as the programming emphasis is placed on the object, not the static flow of control. This is considered further in Chapter 14. For simulation purposes, the first two are far more important than the third, and these are not limited to object-oriented languages. The basic facilities for co-routines and communication can be found in a number of recently developed general purpose high-level languages, particularly Modula-2 and Ada.

12.4 PROCESS DESCRIPTION WITH PASCAL

Process description packages have been written in Pascal. PASSIM (Uyenso and Vaessen 1980), for instance, is a fairly complete implementation of GPSS in Pascal. This section will show how Pascal_SIM can be used to write process description simulations.

Each process is written as a procedure containing a *case* statement where each arm of the *case* statement is a block. The simulation performs the procedure while the entity is active and leaves it when the entity is suspended. When an entity is reactivated from the calendar or suspended chain by calling the procedure, a variable indicates the next block that it

should attempt. Using Pascal_SIM, the next block can be specified by the **next_b** field in the entity record, i.e., **next_b** now stands for *next block to be attempted* rather than *next bound event*. A block may correspond to a bound or conditional event, or a number of combined bound events. If each entity taken off the calendar and every entity on the suspended chain are made **current** prior to attempted reactivation, then a process can have the form:

```
procedure <process name>;
  begin
    case current^.next_b of
      begin
        1: < first block >;
        2: < second block >;
           :
        etc.
      end;
  end < process description >;
```

In GPSS, the executive can work out when an entity is blocked (i.e., cannot continue to the next block of the process until some condition has been specified) and add it to the suspended chain. Using Pascal_SIM, it is necessary to establish this within the procedure. This can be done by using a *repeat...until* loop and a flag, using, for instance, a Boolean *finished*, thus:

```
procedure <process name>;
  var
    finished :boolean;
  begin
    finished := false;
    repeat
      case current^.next_b of
        1: < first block >;
        2: < second block >;
           :
        etc.
      end;
    until finished;
  end < process description >;
```

One block can move an entity to the next block by altering its **next_b** value and leaving *finished* as false or it can suspend it by setting finished to *true*. The variable *finished* must be altered within the arm of the *case* statement that represents the block. The theatre process description does not need a *finished* flag because the theatre never moves through a number of blocks.

Some blocks can only be completed if sufficient resources are available. These blocks correspond to the conditional events of the three-phase approach. The procedure **cause** is used where *advance* would be used in GPSS. The first parameter of the **cause** statement is now a block rather than a bound event number.

Appendix 12.A gives detailed pseudo-code for a process description version of the hospital simulation, using these ideas. To make process descriptions slightly more readable, two procedures are provided in Pascal_SIM specifically for process description simulations. The procedure:

procedure **branch** (*next* :**cardinal**)

can be used to move an entity to any other block in the *case* statement; so **branch** is the equivalent of *transfer* in GPSS. Rather than use **dis_entity** to terminate an entity, a procedure **remove_entity** is provided. This signals to the executive that the entity has finished its process, and in disposing of it, the executive ensures that the entity is not left on the suspended chain.

The complete program for the hospital simulation using the process description world view is shown in Appendix 12.B. The executive is an implementation of the time-flow mechanism algorithm presented in section 12.2.1. Whereas with the two- and three-phase approaches the programmer must enter the names of the bound and conditional events in the executive, here the names of the procedures representing the process descriptions must be entered. The class number of an entity is used to relate it to the correct process.

The Pascal_SIM process description program for the simulation is fairly ungainly code and much of the easy comprehension that is evident in GPSS is not apparent. The approach to process description in Pascal_SIM is intended to be illustrative rather than definitive and could be much improved. For instance:

(*a*) procedures that both check availability of resources and allocate them to entities could be developed, removing the need to use checks of the type *if* < resource available > *then* < allocate > in the process descriptions;
(*b*) priorities could be added to the entity record, and initialized in **new_entity** so that the executive could then be extended to maintain the suspended chain in priority order.

The program shown in Appendix 12.B collects no statistics and produces no visual output. Both of these are more difficult when using a process-oriented view because the queues for resources are not available for

counting or display. GPSS overcomes this problem by providing special queue statistics blocks, in particular *queue* and *depart*, where an entity is time stamped when it encounters a *queue* block. It then has the total time it spent between the *queue* and *depart* blocks added to statistics counts. Similar procedures could be added to Pascal_SIM and used with the histogram facilities.

12.5 WHICH WORLD VIEW?

All three world views are widely used. However, the majority of simulation programmers tend to use the same approach for all modelling. Often their choice is based solely upon their past education or fashion (event scheduling has predominated in the USA, whereas the three-phase approach has been common in the UK). However, recently it has been realized that each world view has its uses and many specialized simulation programming languages now provide more than one world view. For instance, SIMSCRIPT II.5 (Russell 1983) provides both process description and event scheduling, and SLAM II (Pritsker 1984) provides all three (although in both SIMSCRIPT II.5 and SLAM II, process description is preferred).

12.5.1 Process view

For simple queueing simulations, where entities queue for services constrained by resources, process description is probably best. It is easy to comprehend and puts the emphasis on modelling the dominant entities in the simulation. The programmer need not be concerned with arranging for entities to join, and to be taken from, queues.

Process interaction is very powerful for modelling systems which are message based, where entities send messages and act on messages received. Problems arise when different types of entities compete for resources. Priority for access to resources by entities, when handled by assigning priorities to different entity classes (as discussed in Chapter 7), is sufficient for many applications. However, the complex assignment of resources to entities can be difficult to model (and in some cases impossible), because the programmer does not have easy access to the queues (which are subsumed into the suspended events chain). Furthermore, the queues are not readily available for incorporation into the visual display, such as those discussed in Chapter 9. The techniques discussed in Chapter 11, such as descheduling and shadow entities, are also difficult to program within the process view.

12.5.2 Three-phase approach

The three-phase approach is most appropriate for modelling the complex assignment of resources. Any *decision rules* regarding engagement are collected into one place, namely the appropriate conditional event. As queues are handled explicitly, their manipulation is relatively easy. It is excellent for providing simulations with visual output, because the screen can be updated after scheduled occurrences (i.e., within bound events), after conditional occurrences (i.e., within conditional events), or after an entire time beat.

The main disadvantage of the three-phase approach is its relative inefficiency. As the size of a model increases, the number of conditional events increases and, thus, the number of *wasted* calls to conditional events (where no action can take place) increases. This can be overcome by cellular simulation, as described in Chapter 11, but this increases the burden on the programmer. However, process-based approaches can also be inefficient, because the size of the chain of suspended events increases as the size of the model increases.

12.5.3 Event method

Event scheduling avoids the problems of computing inefficiency by placing on the programmers the burden of working out when conditional actions can take place. The resulting simulation is thus more efficient, but less modular.

12.6 SUMMARY

Process description, as evident in GPSS, and process interaction are two types of process view, the third major discrete-event world view. By using *case* statements to represent processes, process description models can be developed in Pascal_SIM. However, the structure of the executive is rather more complex than for the other world views. Process interaction is distinguished from process description by the provision of communication facilities, a co-routine facility, and inheritance.

The choice of a world view is confused by the terminology and the subtle differences of approach; yet each has its advantages:

(*a*) the three-phase approach is the most flexible approach for modelling systems where there is complex interaction between entities and resources and for writing visual simulations;

(*b*) the process view may be the easiest approach for describing a straightforward queueing system with simple priorities for the use of resources;
(*c*) a process interaction approach is particularly appropriate for message-oriented simulations;
(*d*) the event method should be considered when computing efficiency and the speed of execution is of the utmost importance.

12.7 EXERCISES

1. In using the process description world view in Pascal_SIM, the programmer must check the availability of each resource, schedule the next block that the entity enters and set a flag so that the process is terminated. For example:

```
2: begin
     if bed.num_avail > 0 then
       begin
         cause (3,current,negexp(60,hospital_stay_seed));
         acquire (bed,1);
       end;
   finished := true;
   end;
3: < the next block >
```

Develop new versions of **acquire** and **release** which work like the GPSS blocks SEIZE and RELEASE so that the availability of the resource is checked and the entity is either suspended (when the resource is not available) or passed to the next block (when the resource is available). Thus the above coding might look something like:

```
2: process_acquire (bed,q1);
3: cause (3,current,negexp(60,hospital_stay_seed));
```

2. Using the Pascal_SIM facilities for creating and updating histograms (**make_histogram**, **log_histogram**), produce two new procedures:

```
stat_queue (var e :entity);
```

and

```
stat_depart (h :histogram; e :entity);
```

where **stat_queue** will time stamp the entity with time **tim** (using an additional entity attribute) and will calculate **tim** minus the entity's previous time stamp and add this value to histogram, *h*. (Note: these are equivalent to the GPSS blocks QUEUE and DEPART.)

3. *Hospital System*. Implement the process description version of the simulation (Appendix 12.B). Adapt it to collect the statistics shown in Chapter 6. (Hint: you will have to keep track of the lengths of the different queues.)

4. *Repair Shop System*. Write a process description simulation of this system.
5. Assume that all entities in a queue are in the same class and have the same color. Write a simplified version of **write_queue**, called:

```
write_q (x,y,qlen,max_length :cardinal; cl :class_num; b,f :color);
```

where *cl* is the class number, *f* is the foreground color and *qlen* is the length of the queue.

Use this procedure to write a process description version of the visual hospital system simulation described in Chapter 10. Note that, as in Exercise 3, you will have to keep track of the lengths of the different queues.

* * *

12.A PSEUDO–CODE FOR HOSPITAL SIMULATION USING PROCESS DESCRIPTION

```
procedure patient1;
  var
    finished :boolean;
  begin
    finished := false;
    repeat
      case current^.next_b of
        1: begin
             < cause the arrival of a new patient1 >;
             < move onto block 2 >;
           end;
        2. begin
             if < bed available >
               begin
                 < acquire a bed >;
                 < hospital stay (cause entry to block 3) >;
               end;
             finished := true;
           end;
        3: begin
             < return the bed >;
             finished := true;
             < terminate >;
           end;
      end;
    until finished;
  end < process description for patient1 >;

procedure patient2;
  var
    finished :boolean;
```

```
    begin
      finished := false;
      repeat
        case current^.next_b of
          begin
            1: begin
                 < cause the arrival of a new patient2 >;
                 < move onto block 2 >;
               end;
            2: begin
                 if < bed available > then
                 begin
                 < acquire a bed >;
                 < pre-operative stay (cause entry to block 3) >;
                 end;
                 finished := true;
               end;
            3: begin
                 if < theatre open and available > then
                   < operation (cause entry to block 4) >;
                 finished := true;
               end;
            4: begin
                 < theatre available again >;
                 < post-operative stay (cause entry to block 5) >;
                 finished := true;
               end;
            5: begin
                 < return the bed >;
                 finished := true;
                 < terminate >;
               end;
        end;
      until finished;
    end < process description for patient2 >;

  procedure theatre;
    begin
      case current^.next_b of
        1: begin
             < open theatre >;
             < cause closure (entry to block 2) >;
           end;
        2: begin
             < close theatre >;
             < cause opening (entry to block 1) >;
           end;
      end;
    end < process description for theatre >;
```


12.B HOSPITAL SIMULATION USING PROCESS DESCRIPTION

```
program simulate (output);

  const
    amount_of_beds      = 20;
    patient1_seed       = 1;
    patient2_seed       = 2;
    hospital_stay_seed  = 3;
    pre_op_stay_seed    = 4;
    operation_seed      = 5;
    post_op_stay_seed   = 6;
    theatre_open_time   = 4;
    theatre_closed_time = 20;

  var
    bed      :bin;
    q1,q2,q3 :queue;
    theatre  :record
                body           :entity;
                open,available :boolean;
                { true if theatre is open, available }
              end;

procedure patient1;
  var
    finished :boolean;
  begin
    finished := false;
    repeat
      case current^.next_B of
        1: begin
             cause (1,new_entity(1,1),
                    negexp(6,patient1_seed));
             branch (2);
           end;
        2: begin
             if (bed.num_avail>0) then
               begin
                 cause (3,current,
                        negexp(60,hospital_stay_seed));
                 acquire (bed,1);
               end;
             finished := true;
           end;
        3: begin
             return (bed,1);
             finished := true;
             remove_entity;
           end;
      end;
    until finished;
  end { process for patient 1 };
```

```
procedure patient2;
  var
    finished :boolean;
  begin
    finished := false;
    repeat
      case current^.next_B of
        1: begin
             cause (1,new_entity(2,1),
                    negexp(12,patient2_seed));
             branch (2);
           end;
        2: begin
             if (bed.num_avail>0) then
               begin
                 cause (3,current,
                        negexp(24,pre_op_stay_seed));
                 acquire (bed,1);
               end;
             finished := true;
           end;
        3: begin
             with theatre do
               if open and available then
                 begin
                   available := false;
                   cause (4,current,
                          normal(0.75,0.25,operation_seed));

                 end;
             finished := true;
           end;
        4: begin
             theatre.available := true;
             cause (5,current,
                    negexp (72,post_op_stay_seed));
             finished := true;
           end;
        5: begin
             return (bed,1);
             finished := true;
             remove_entity;
           end;
      end;
    until finished;
  end { process for patient 2 };
```



```
procedure schedule_theatre;
  begin
    with theatre do
      case current^.next_B of
        1: begin
             open := true;
             cause (2,current,theatre_open_time);
           end;
        2: begin
             open := false;
             cause (1,current,theatre_closed_time);
           end;
      end;
  end { schedule theatre };

procedure run (duration :real);
  var
    c       :link;
    changed :boolean;
    e       :entity;
    present :cardinal;
  procedure reactivate;
    begin
      on_calendar := false;
      present := current^.next_B;
      case current^.class of
        1: patient1;
        2: patient2;
        3: schedule_theatre;
      end;
    end { reactivate };
  begin
    running := true;
    repeat
      if calendar = calendar^.next then running := false
        else
          begin
            tim := calendar^.next^.item^.time;
            if duration<tim then running := false
              else
                begin
                  while (calendar<>calendar^.next) and
                        (tim=calendar^.next^.item^.time) do
                    begin
                      calendar_top;
                      reactivate;
                      if current^.next_B = 0 then
                        dis_entity (current)
                        else
                          begin
                            if not on_calendar then
                              give_tail (suspended_chain,
                                         current);
                          end;
                    end;
```

```
                repeat
                  changed := false;
                  c := suspended_chain^.next;
                  while c<>suspended_chain do
                    begin
                      current := c^.item;
                      reactivate;
                      c := c^.next;
                      if on_calendar or
                      (present<>current^.next_B) then
                        begin
                        changed := true;
                        e := take (suspended_chain,c^.pre);
                        end;
                      if current^.next_B = 0 then
                        dis_entity (current)
                        else
                          if  (not on_calendar) and
                          (present<>current^.next_B) then
                            give_tail (suspended_chain,
                                                  current);
                    end
                until not changed;
              end;
          end
      until not running;
    end { run };

  procedure initialize;
    begin
      make_sim;
      make_streams;
      make_bin (bed,amount_of_beds);
      make_queue (q1); make_queue (q2); make_queue (q3);
      with theatre do   { create theatre }
        begin
          body := new_entity (3,1);
          open := true; available := true;
          cause (1,body,theatre_closed_time);
        end;
    end { initialize };

  procedure report;
    begin
    end { report };

    begin
      initialize;
      cause (1,new_entity(1,1),0);
      cause (1,new_entity(2,1),0);
      run (24*30*12);
      report;
    end.
```

13

Using Simulation to Make Decisions

The main purpose of building simulation models is to provide more information for decision-making by management, unions, or whoever has commissioned the simulation. However, in the process of building the model, of collecting and analysing results, it is easy to lose sight of the purpose for which it was designed.

This chapter will show how to simplify our view of the model and to look at it in the context of its environment. In particular, we discuss the need for clear objectives, credible models, reliable input and pertinent output.

13.1 MODELS IN DECISION-MAKING PROCESS

13.1.1 One-off models

Recent techniques with menus, visual and natural language interfaces, such as those described in the next chapter, make models of all kinds much easier to develop and use. This, together with the availability of microcomputer technology, has made mathematical and simulation models much more accessible to the users.

Therefore, managers are increasingly likely to use models for making small *one-off* decisions as well as for large and important ones. Models might, for example, be helpful in the following types of problem:

(*a*) whether and which new equipment should be bought,
(*b*) whether and how to rearrange customer facilities (such as service tills in a bank),
(*c*) to decide if it is better to use the available equipment to make one product line rather than another.

Managers responsible for making the decision are likely to be very interested in the results produced by the model and, in general, will want to

be involved closely in the model's development and validation. It is important, therefore, that the model should be credible and easy for them to use.

The data should be reliable and accurate but, because the model is unlikely to be needed again, much of the data collection can be of an *ad hoc* nature.

13.1.2 Models for long-term use

In most organizations there are classes of decision that have to be made time and time again. Some examples are:

(*a*) decisions on budget allocations, based on past performance and projections for the future;
(*b*) the management of road traffic within a particular area covering one-way systems, traffic light timing, and major road works;
(*c*) manpower planning.

These models need to be flexible and robust to changes in input data, activities, and policies within the organization. There is clearly a need for accurate and up-to-date input data. Where models are integrated with the data collection and information provision activities of the organization, the whole system is called a *decision support system*. For more information in this area readers should refer to the book by Keen and Scott-Morton (1978).

13.1.3 Cost models

Clearly, costs are a very important influence in decision-making. For some simulations, it may be possible to summarize all the output measures in terms of costs but it is not always feasible. Whereas the cost of providing the salary of a person to work in a given role is clear cut, the cost of maintaining long queues in a system is, inevitably, approximate. Nevertheless, such cost information as is available is always of interest in any organization.

Costs are dependent on an underlying cost model. The resources, while they remain constant in a simulation program, provide the basis for estimating some of the fixed costs while the activities give rise to the variable costs. For example, in the hospital system the costs may be allocated as follows.

(*a*) A ward may be allocated staff depending on the number of beds, and the theatre allocated staff while it is open and available. Therefore, these staffing costs may be regarded as fixed, varying only with the resource levels.

(*b*) The variable costs of treatment, drugs, and pathology tests, will vary directly with the number of patients taking part in the activities in the system.
(*c*) The overheads, such as building maintenance and administration, may be regarded as being completely fixed, varying with none of the simulation factor levels.

These assumptions form the basis of a *cost model* which is related to, or integrated with, the simulation model. There is, inevitably, a difference of opinion about the correct balance between fixed and variable costs for any particular process. It could be argued, for example, that nurses will not be provided for an open but empty ward and therefore they should not be related to the number of beds. Where there is doubt, it is perhaps best to be guided by the current actuarial practice of the organization for whom the work is being done.

The cost model may either be incorporated in the structure and logic of the simulation program or else may be applied to the model output. Financial planning packages and spread sheets are extremely convenient for performing cost calculations on model output. However, the most satisfactory approach for any problem will depend on the circumstances and the decisions that need to be made.

13.2 ADVANTAGES OF SIMULATION MODELLING

Previous chapters have emphasized that, after identifying the problem to be solved, the first step in building a model is to define the objectives and derive assumptions. In theory, the choice of modelling technique should follow from this. In practice, most professional management scientists have a strong preference for one type or class of technique and will decide on the modelling approach at a very early stage in the project. In any case, inevitably, the choice of technique will affect the way in which the objectives are worded, how tightly they are defined, and the assumptions that are made.

13.2.1 Defining objectives and making assumptions

Chapter 1 pointed out that simulation is a descriptive modelling technique which is effective for portraying stochastic systems. Unlike the prescriptive deterministic modelling techniques, such as linear programming, the mechanics of simulation do not require the definition and solution of an objective function. However, a simulation project does require clearly

worded objectives in order to determine the model's structure, input and output (see section 13.3).

All mathematical and simulation models are based on assumptions. Some of these are derived from the objectives and are natural simplifications of the system while others are dictated by the chosen modelling technique. Mathematical models require much more restrictive assumptions than simulation models. Davies (1985a), for example, describes assumptions that have to be made for analytical stochastic models but not for simulation models. These include:

(*a*) arrival and service time distributions have to fit the mould of well-known parametric distributions,
(*b*) individual entity characteristics cannot influence their progress through the system,
(*c*) events are assumed to happen at discrete and equally spaced points in time.

There is a danger that, because of their flexibility, simulation models are made much too complicated. However, this can be avoided if they are built up from simple to more complex structures in the way described in Chapter 8.

13.2.2 Credibility

In order to have confidence in a model, those using it should not only be able to appreciate the model assumptions and their implications, but also understand in some detail how the model works. They are then able to check to their satisfaction that the model behaves in all important respects as the system does in practice. This is particularly important for one-off models where management staff are likely to want to be involved in its development.

The simple deterministic models, the simple stochastic models (such as the Markov models), and synchronous simulations (which move forward in equal time slices) are easy to understand but often system descriptions are pressed into the mould of unsuitable techniques. These models then lose credibility because of their poor reflection of system properties. Furthermore, the greater the effort that is made to develop analytic models to match the system and meet assumptions, the more complex and difficult to understand they become.

The great advantage of discrete-event simulation is that it can follow very closely the logic and flow of the system being modelled. If it reflects system properties well and is written or documented in a way that is easy for the user to understand, then it should be very 'credible'.

13.2.3 Robustness

If a model is used at different points in time or at other locations from where it was developed originally, changes to the input data or to the structure may be needed. Robustness is particularly important for models which are to be used on a long term basis.

Changing the logic and introducing complexities into mathematical models is difficult and, often, impossible. However, simulations are much more flexible. Chapters 7 and 11 have described techniques for changing the internal structure and logic of a simulation model. The three-phase approach, in particular, is very robust to changes (Crookes 1982).

13.2.4 Ease of use

The portability of microcomputers has facilitated the use of models by the decision-makers themselves and therefore it is important that models should be easy to use. Although ease of use relates to the implementation of the modelling method rather than the method itself, simulation models with independent entry of distribution data and with the possibility of interactive facilities are particularly appropriate.

13.3 TYPES OF SIMULATION MODEL

Simulation modelling is a powerful and flexible technique which facilitates the development of credible and robust models. Its adaptability has certainly given rise to a wide range of applications. Chapter 1 distinguished three types of simulation model each of which play a different role in the decision-making process.

13.3.1 Predictive simulations

Predictive simulations are used to determine average results and confidence limits of a simulation run which has specific factor levels. Although the factor levels may be changed, interest will focus on absolute outcome measures rather than on comparative ones. Predictive simulations are usually most appropriate for simulations which take some time to reach a steady state. Several runs will invariably be needed, therefore, in order to obtain accurate estimates of results. The accuracy of these results may well be increased by variance-reduction methods. Time series and regression techniques are useful for analysing results.

This type of approach has been used to predict numbers of patients needing treatment for irreversible renal failure (Davies 1985b) where it is more appropriate than traditional forecasting techniques because the simulation can describe the increasing demand for treatment, improving survival rates, and the complex way in which treatments are allocated and changed.

Models of this type are likely to have specific and well-defined objectives and are likely to be needed on a long-term basis. They may thus be integrated in a decision-support system. Reliable current and historical input data will certainly be needed for accurate forecasts.

13.3.2 Comparative simulations

Simulations are often used to determine whether one option is 'better' than another. For example, it may be important to determine whether a one-queue system in a bank is better than the traditional system of having a queue for each server.

Vague objectives like this have to be made more specific so that it is possible to define appropriate output measures. It is often thought by those who are in the habit of waiting in queues, rather than managing them, that the most important objective is to keep queues as short as possible. 'Better' would therefore mean 'has shorter queues'. However, there is a trade-off between server time and queue length, and servers invariably cost the management money. Nevertheless, long queues block up systems, discourage custom, and usually require space to accommodate them. If the objectives cannot be made specific at the outset, the simulation can provide a range of output measures. However, this only delays the time when the decision-maker must decide on their relative importance.

Where the output measures are clear, statistical hypothesis techniques may be used to analyse results (see Chapter 5). This type of model may be used on a long term or one-off basis and, in either case, it will need accurate input data to provide reliable output for decision-making.

13.3.3 Investigative simulations

An investigative simulation should indicate the major factors which affect the flow of entities in a system but it is not required to provide precise answers. Therefore, the quality of the input data is of less importance than in the previous examples. Visual interactive techniques are usually extremely suitable, enabling management to explore the effects of changing factor levels while a simulation is running. Such simulations should provide a good understanding of system characteristics which is necessary for informed decision-making.

When more accurate answers are required, simulation results may be collected in histograms or analysed statistically. However, Chapter showed that it is difficult to repeat and compare results from *interactive* simulation runs.

13.4 CLARIFYING OBJECTIVES

13.4.1 Simulation model as a black box

Sometimes, in order to consider a model as part of a decision-making process, it is helpful to close your mind to the complicated internal logic of the model. It is thus regarded as a black box. Figure 13.1 shows a simulation as a black box together with the inputs to and the outputs from the model. The inputs and outputs are for a general queueing model and are based on those discussed Chapters 4 and 5.

In any particular system, the output arrows showing the simulation results are derived directly from the objectives and may differ considerably from one problem to another. The input arrows are more closely related to the model assumptions and, in a simple model, only a subset of those shown in Fig. 13.1 may be needed.

In Chapter 6, the objectives of the hospital system are: 'to investigate the effect of bed and operating theatre provision on patient waiting times'. Figure 13.2 shows the output measurements which are derived directly from the objectives. The input side of the diagram includes all the factors that may be varied in this problem. The decision variables are derived from the objectives and are underlined. Factors which are assumed to have no

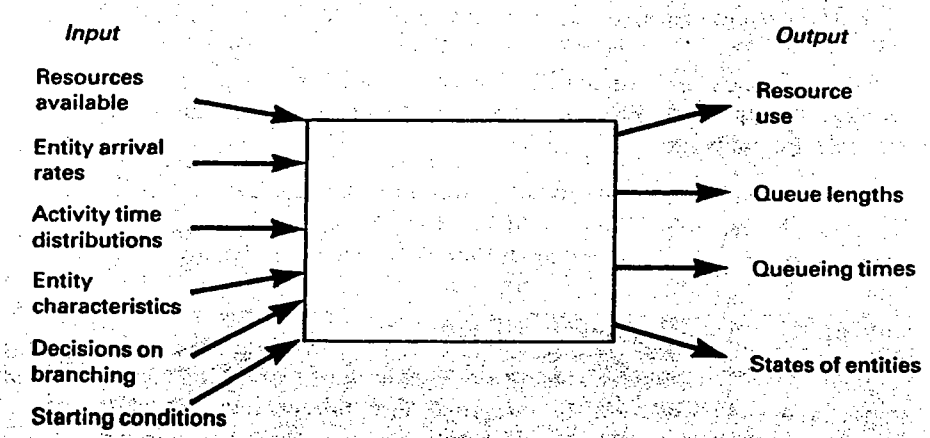

Figure 13.1 Input–output diagram for general queueing system

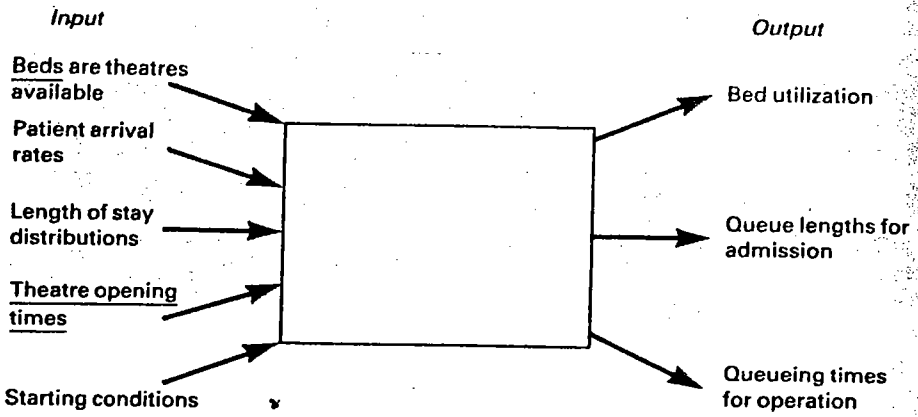

Figure 13.2 Input–output diagram showing factors, decision variables, and measurements for hospital system described in Chapter 6 with decision variables underlined

effect on the model output (such as entity characteristics) are omitted from the diagram.

Input–output diagrams are thus very simple modelling aids which are helpful in the following ways:

(*a*) to identify input data requirements and decision variables,
(*b*) to provide information on data collection requirements for validation,
(*c*) to determine the output requirements of the model.

13.4.2 Difficult Objectives

Often objectives are phrased in a way that appears inappropriate for a simulation approach. For example, an objective in the machine repair example might be to:

(*a*) provide sufficient repair staff to enable at least 10 machines to be working for 90 % of the time, or
(*b*) provide the number of repair staff that will minimize the cost of lost production,

and in the hospital example the objective might be:

(*c*) given that we have a certain allocation of money to run this ward and theatre, determine the best number of beds and theatre sessions to treat the most patients.

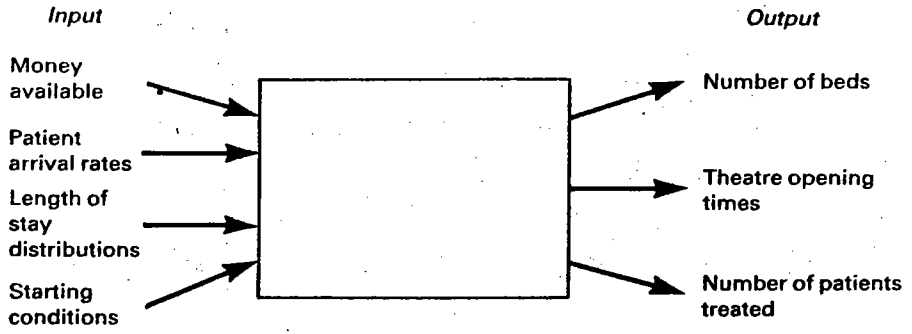

Figure 13.3 Input–output diagram of hospital system where objectives are: 'given that we have a certain allocation of money to run this ward and theatre, determine the best number of beds and theatre sessions to treat the most patients'

In all these examples, some of the normal inputs become outputs and vice versa. Some of the arrows on the input–output diagram are thus reversed. In the first example, server utilization, which is normally an output, has become an input. The second two examples involve costs and a requirement for an optimization approach. Figure 13.3 shows the input–output diagram for the last one.

The discrete-event simulation techniques that have been discussed are not very appropriate for optimizing problems. However, an experimental approach may be adopted, by 'trying' different staffing or resource levels until one is found that appears to meet the criteria. Resource levels at around this level can then be explored more closely wih different random number streams. Using this approach, we are turning the arrows in the input–output diagram back round again so that resources or staffing levels are the input, and server busy time or workload and the related costs become the output. The next chapter will discuss some prescriptive simulation, techniques which can sometimes be used for this type of problem.

13.5 PROVIDING SIMULATION INPUT AND OUTPUT

Chapter 4 described the difficulties in collecting data for simulation models. One-off models may use estimated values and the results of special surveys. However, when models are for long term use the data must be kept up-to-date and be available when required. Any factors which may need to be changed at any time must be accessible to those running the program while the starting conditions need to be read in quickly and easily.

Statistical and visual output is discussed at length in Chapters 5 and 9.

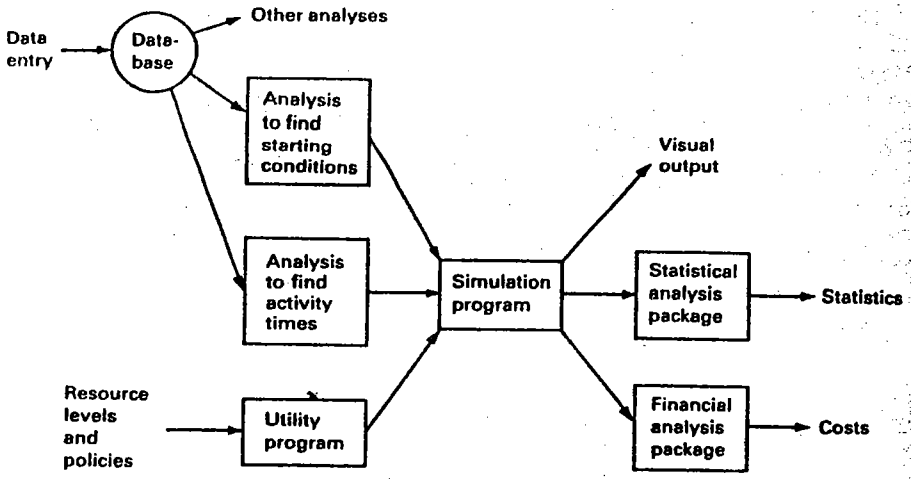

Figure 13.4 Simulation program as part of decision support system where entity data is held on a database; visual output, statistical analyses, and costs are provided

Statistical information may be analysed by a statistical computer package. Costs, which are a particularly useful output measure, require the integration of a cost model and may be analysed with a financial package or spread sheet.

Figure 13.4 shows a simulation as part of a decision support system. The database holds entity information and provides the simulation with starting conditions and activity times. Decision variables such as resource levels and current operating policies are supplied separately. Although this simulation provides visual output, results are also fed to statistical and financial packages for detailed analysis.

13.5.1 Decision variables

Any utility program offering the ability to change decision variables should be provided with default values and be robust to user errors. If the program is screen driven, users should have the opportunity to go back and change screens if they do not like the look of what they have written. The same variables may need to be changed several times during the course of an interactive simulation.

The most common factors that may need to be changed in a simulation program are as follows:

(*a*) resource availability, disponibilidad
(*b*) arrival rates and service times,

(*c*) probabilities of taking different decision paths within the simulation.

Resource availability, such as the number of hospital beds or the number of repair staff for the machine breakdowns, is an exact integer value which can be chosen from a prescribed range. Scheduled resources, such as operating theatre timetables, are more difficult to present on a computer screen.

Where interarrival rates or service rates are described by parametric distributions, the parameters, usually the distribution means, are the factors of interest. These can be varied quite easily. Non-parametric distributions cannot easily be varied in a systematic way and any change will usually require a completely new set of histogram values.

13.5.2 Historical data

If the simulation is to be used on a long-term basis, factors other than the decision variables will almost certainly need to be changed at some time. For example, operating practices and staff changes in the organization affect decision paths and activity time distributions. Therefore, to avoid amending and recompiling programs, it is advisable to keep the data in disk files and to have separate programs for updating the data and reading them into the simulation program.

13.5.3 Starting conditions

Some simulation programs, particularly those used for forecasting purposes, need 'up-to-date' information to set the simulation starting conditions. Where these consist of simple counts of entities in different states, they can be entered in the simulation in the same way as the decision variables. If, however, they require the entry of individual entity data showing their characteristics or history, then it is more useful to maintain this information on a database than to enter it by hand on each occasion.

In order to use the entity information in the simulation to set the starting conditions, each entity must be put either in a list or else in the calendar together with information about the time and number of its next bound or scheduled event.

13.5.4 Databases

Databases can be used to provide information about the current state of entities, activity times and decisions within the organization. Some databases have query languages which interface with Pascal. Therefore, a

simulation program in Pascal could call directly on the database for input information. Where this facility is not available, then the database query languages must be used to write files of data which can be read by the simulation program. Much of the data entry for the simulation program is thus completely automated.

Even though the data on the database may be accurate, it is important to introduce additional validation checks. It is necessary, for example, to check that there are sufficient data points to derive the distribution parameter estimates and for the production of histograms based on empirical distributions. It is no longer possible to do these on the *ad hoc* basis which is adequate for most one-off simulations.

13.6 SUMMARY

Simulation is a very powerful technique which can be used for many different types of problem. *Clear objectives* enable decision variables and output measures to be determined. However, some simulations may be written simply to understand how a system works. Simulations which produce graphical and visual output are useful for meeting these rather vague objectives and help the user to form more precise objectives.

Simulations which are used on a *long-term* rather than a *one-off* basis need reliable and regularly updated input and output which is comprehensive and flexible. A cost model may need to be incorporated in the simulation model to provide a unique measure of performance or profit.

13.7 EXERCISES

1. Ms Smalltime buys a certain model of computer for £5778 and sells it for £7 500. She makes an additional profit of approximately £300 on software, on each computer sale. She buys the computers on borrowed money which costs her approximately £20 per computer per week. Computers may be ordered each week but there is a minimum order size of five computers. It takes three weeks to receive a new shipment of computers after an order is placed. The cost of placing an order is £50. If there is a demand for a computer when she has run out of stock she loses the sale of, not only the computer, but also the additional software.

   Suppose she has 20 computers in stock. Her current re-order policy is to order 15 computers when the inventory has fallen to 10 or fewer. The arrival of customers requiring this type of computer follows a Poisson distribution with a mean of 2·84 customers per week. Ms Smalltime wants you to recommend the best ordering policy.

   (*a*) Write down the objectives and draw an input–output diagram. If you are to use discrete-event simulation to model this system, show how you will use this descriptive technique to meet the objectives.
   (*b*) Design an experimentation plan. Write and run the simulation to solve the problem.
   (*c*) List the model assumptions and limitations.

2. *Bank System*. Using the case study description in Chapter 1 (Appendix 1.A.1) and the work done in the exercises at the end of previous chapters, design and write a simulation program which the bank manager can validate and experiment with.

3. *Traffic Light System* (Appendix 1.A.2). Discuss the problems inherent in using a simulation model to predict the traffic flow on a long-term basis. Suggest practical solutions to these problems.

14

Advances in Computing and Simulation

The major advantage of using Pascal_SIM for writing simulation programs is that it employs a commonly used structured high-level language. It is thus easy to learn for those with some Pascal experience, is very portable, and extremely adaptable. However, the ambitious modeller may wish to describe structures, provide or use interfaces and tools, or use techniques that are not readily available in Pascal or Pascal_SIM. These include:

(*a*) a separation of the instructions for the visual output from the simulation logic, so as to keep the program modules small and to allow ease of maintenance;
(*b*) more realistic modelling of the decision mechanisms of entities, rather than depending solely on probabilities determined by prior observation and sampling;
(*c*) the development of simulations which automatically optimize results;
(*d*) the use of an interface which makes it possible for the user to describe and model systems without having to program it in Pascal or any other language;
(*e*) the use of fast and efficient simulation packages tailored to specific industries or institutions;
(*f*) use of a total computer system or environment for simulation modelling, including data input, analysis and presentation of results.

This chapter describes how advances in computing and simulation are making these facilities, among others, available to the modeller.

14.1 PROGRAMMING LANGUAGES

14.1.1 Simulation programming languages

Many of the more advanced techniques described in this chapter are built on established simulation programming languages (SPLs). Popular SPLs

include: ECSL (Clementson 1978), SIMSCRIPT (Russell 1983), GPSS and SIMULA (see Chapter 12). The advantage of using an SPL, rather than a high-level language, is that:

(*a*) most provide comprehensive facilities for sampling, list control, and entity definitions;
(*b*) the simulation executive is hidden from the user and never needs to be altered;
(*c*) most provide good report-writing facilities.

However, the disadvantages are that:

(*a*) an implementation may not be available on a particular machine;
(*b*) they lack flexibility – it can be impossible to provide for user interaction or for more complex structures (for example, the shadow entities described in Chapter 11);
(*c*) run-time efficiency can be very poor;
(*d*) the limitations of a commercial package may not become apparent for some time, whereas the use of a tool-kit of facilities in a high-level language allows complete flexibility and adaptability.

More advanced facilities are now being built on some SPLs. For example, SLAM II is at the heart of the TESS environment (see section 14.6 below) and recent versions of SIMSCRIPT provide animation (see section 14.2).

14.1.2 Newer general purpose languages

An attractive alternative to using an SPL or a set of facilities such as Pascal_SIM, is to use a tool-kit developed in a newer general purpose language. Candidates include Wirth's successor to Pascal, entitled Modula-2 (Wirth 1985), and Ada.
Both Modula-2 and Ada have two advantages over Pascal.

(*a*) Both allow the programming of interacting *processes*. Thus a true process interaction package can be developed.
(*b*) All the provided facilities can be collected into one place, called a *module* in Modula-2 and a *package* in Ada. Data shared by the simulation routines can be hidden from the simulation programmer. In Ada, a *generic package* can be defined, such that an instance of it can be generated with specific data structures. Thus a simulation package does not have to be redefined each time the entity type changes, as with Pascal_SIM.

There are a number of discrete-event simulation packages for Ada, for example, see the work of Bryant (1982).

14.2 ANIMATION

Chapter 10 showed how to provide simple visual displays in Pascal_SIM by coding individual statements within each event. However, these statements confuse the simulation logic and greatly increase the length of programs. An alternative approach is to design an animator which is driven by output from the simulation logic.

Figure 14.1 shows that the modeller or developer designs the picture and writes the simulation as two separate exercises. He uses a software tool to design the picture, usually employing a high quality graphics terminal. In designing the static background, he must select icons to represent the entities (perhaps designing new ones as required) and position the queues or activities in the picture. The simulation program is written as if there were no visual output.

When the simulation runs, it produces formatted output which is decoded by the animator. Consider, for example, the simulation logic which states that a particular entity finishes an activity and is put on the end of a queue. The output produced by the running simulation, coupled with the information for the separately designed display, enables the animator to identify the position of the activity and queue on the screen and move an icon representing the entity from one to the other.

This approach has the considerable advantage that the formatted output can be saved to file and run independently of the simulation program. However, animation is much less flexible than the direct programming method described in Chapter 9. It will only cope with a standard range of pictures and simulation logic.

Animators are normally designed to work with a specific SPL or package. Commercial examples of animation systems include CINEMA

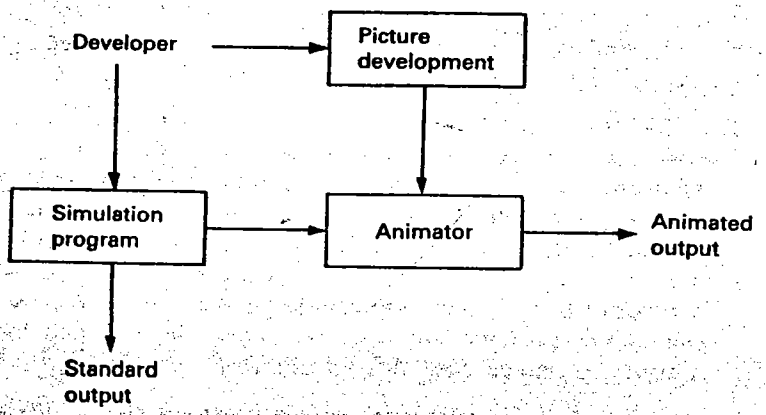

Figure 14.1 Interaction between developer and animator

(Pegden *et al.* 1985) which is an animator for simulations written in the SIMAN language, and the facilities of the TESS environment (Standridge 1985) which animates simulations written in SLAM II. Others have been developed in-house for specific purposes.

14.3 KNOWLEDGE REPRESENTATION

Chapter 7 showed that entities may be given characteristics that influence their progress through the system. Activity time distributions, for example, may be related to the attributes of an entity, or branching probabilities may depend on the previous activities that the entity completed.

However, in modelling some systems, such facilities are inadequate. Entities can take autonomous decisions, based on some complex decision-making mechanism. For example, a manufacturing simulation may describe a robot cell whose choice of task sequence is dependent upon available materials, the production schedule, a scheduled maintenance, and other factors.

One approach to modelling complex decision-making mechanisms in simulation is to use the methods of *knowledge representation* developed in *artificial intelligence* (AI). Knowledge representation is concerned with describing information about a system, or the details of a system, as a computational structure. Within AI systems, this knowledge can then be used by a computer program, for example, to provide answers to questions, guidance to users or suggest courses of action. Knowledge differs from simple factual data that might be stored in a database, in that the structure may be variable, dynamic, and may indicate degrees of uncertainty about the knowledge.

14.3.1 Rule-based approaches

The commonest knowledge representation method is production of situation–action rules, where chunks of knowledge are expressed in the form:

$$\textit{if } \{s_1,s_2,s_3,\ldots,s_j\} \textit{ then } \{c_1,c_2,\ldots,c_k\}$$

where $s_1,\ldots,s_j$ are facts, or conclusions from other rules, and can be combined with the logical operators *and*, *or*, and *not*. If the result of the logical operation is *true*, we can conclude that $c_1,\ldots,c_k$ are true. Frequently, for reasons of implementation, it is useful to use only rules that have a single conclusion.

In simulation, conditional events can be considered as situation–action rules. Consider the hospital system, introduced in Chapter 1 and

programmed in Chapter 3. The conditional event that starts a hospital stay is:

if < not (queue empty) >
  and < a bed is available >
then < admit the patient >
  < start a hospital stay >;

Using the three-phase approach, it is quite straightforward to program a simulation in a rule-oriented language (for example, see Bruno *et al.* 1986, on programming a simulation in OPS5).

More importantly, complex decision mechanisms can be expressed as a set of rules. Rather than using simple queueing priority rules, or using a single probability to express behavior following an activity, a set of rules can be developed. The course of action that an entity takes can then be determined by access to this rule-base.

14.3.2 Object-oriented approaches

An alternative to rule-based knowledge representation is object-oriented programming. In an object-oriented representation, there is a one-to-one correspondence between items in the representation system, and items in the 'real' system. The first object-oriented programming language was the simulation programming language SIMULA (Birtwistle *et al.* 1979), which was briefly discussed in Chapter 12. More recent approaches to object-oriented programming include the following.

(*a*) *Frame-based representation* (Minsky 1977), where knowledge is stored as a hierarchy of frames. Each frame is a 'record' of information about an item, or class of items, and can inherit attributes from other frames. Thus each patient will have an individual height and weight, but will inherit attributes about the illness they possess from a frame that represents that illness.

(*b*) *Smalltalk* (Goldberg and Robson 1983), where items are represented as objects that interact by sending and receiving messages. Each object includes the details, called 'methods', of what to do when a particular message is received. Objects can inherit attributes and methods from other objects.

If a simulation package is developed in an AI language that is object-oriented, then the simulation developer can take advantage of any facilities of the language or the environment in which it runs. For example, the following apply.

(*a*) Many object-oriented AI languages allow rules to be attached to objects. Thus rules can be used to model complexity, as discussed above.

(*b*) Animations can be developed separately using graphical objects which can be activated by message-passing.

(*c*) Classes of entities can be developed for a number of different simulations; each individual simulation can inherit the attribute of the class, yet modify it as necessary.

Thus, a number of simulation packages have been developed using object-oriented AI. Examples include SimKit, written in the knowledge engineering environment (KEE) and developed at Intellicorp (Fraught 1986), the Rand Corporation's ROSS (McArthur *et al.* 1986), and the knowledge based simulation system (KBS) from Carnegie–Mellon (Reddy *et al.* 1986).

There have been a number of attempts to produce extensions to Pascal that include some elements of object-oriented programming. For example, SIMONE (Kaubish *et al.* 1976) employs some of the concepts found in SIMULA.

14.4 PRESCRIPTIVE SIMULATION

Normally, a simulation is used as a descriptive modelling tool. Chapter 13 showed that the modelling objectives enable the modeller to determine the simulation structure, assumptions, output analysis, and experimentation. The objectives are not usually used directly by the simulation program. However, where experimentation has very clear and specific objectives it would seem advisable that the simulation runs should move automatically towards achieving them. Such a simulation is called *prescriptive*.

The simplest way of achieving a type of prescriptive simulation is by producing and programming a method of *automatic analysis*. The earliest attempts used heuristic methods, such as surface response techniques. Here an objective is specified for the system under study as a function of measurable outcomes (for example, utilization rates, queue lengths, etc.), and the simulation is run with different parameters under the control of a monitor which attempts to maximize or minimize this function. (Some of the necessary mathematics can be found in Law and Kelton (1982). However, such methods have been little used, largely because the expression of an objective as a function is usually very difficult, and frequently impossible.

An alternative to automatic analysis is *goal-directed simulation*, where an overall goal is specified for the study, and the controlling monitor tries to produce a design that best satisfies that goal. This heuristic approach differs from the automatic analysis approach in that, at least

in theory, a goal can be symbolic (for example, 'don't keep the customers waiting too long') but, in practice, such goals must often be reduced to a measurable quantity (for example, how long is 'too long' ?). Goal-directed simulations require the use of an appropriate symbolic language, such as Prolog. For example, T-Prolog (Futo and Szeredi 1982) is a simple simulation system that allows some goal-oriented programming.

14.5 DEVELOPMENT TOOLS

Programming a simulation in Pascal can be time-consuming and difficult, particularly for the inexperienced programmer. Developers are turning to more sophisticated development tools that reduce, or even remove, the need to do any programming.

14.5.1 Program generators

With a program generator, such as CAPS (Clementson 1978) or DRAFT (Mathewson 1984), the user undertakes a highly structured dialog with the generator, at the end of which a program is generated in a simulation programming language. For instance, CAPS generates programs in ECSL, whereas DRAFT can generate programs in a number of languages. The Pascal based package eLSE (Crookes *et al.* 1986) includes a program generator which generates three-phase simulations in Pascal. Typically, the dialog with the generator is based on a formal or semi-formal description of the simulation model such as the activity-cycle diagram.

Although program generators greatly reduce the initial programming effort, they demand a linear thought process which experienced programmers may find rather tedious (although this is probably a function of the present generators, and not necessarily a function of the method). More importantly, the system to be programmed has to be described in the structured way required by the method. This is inevitably inflexible, and thus the developer has frequently to extend the generated code so as to produce the desired model.

The major advantage is that this method will produce a working simulation program which needs to be validated but not verified and can be extended, if necessary.

14.5.2 Descriptive tools

Descriptive tools, such as HOCUS (Poole and Szymankiewicz 1977) and Inter_SIM (O'Keefe 1987), are similar to program generators, with two

important exceptions. First, no program is generated, and the description is immediately interpreted (for example, this happens with Inter_SIM), or is converted to some immediately executable form (this happens with HOCUS). Second, the model is described by some semi-formal method of specification, rather than following a dialog. For example, in Inter_SIM the user specifies all the activities, resources, classes and queues, together with any necessary data (e.g., activity duration distributions, arrival rates) in a menu environment similar to a spreadsheet.

Descriptive tools are generally easier to use than program generators and where the description is interpreted, interaction with the running simulation can be very natural. They have the disadvantage that, unlike generators, there is no program available for alteration. Most systems allow extensions to be programmed in a common procedural language, normally FORTRAN or Pascal, and for these extensions to be used from within the method of specification. For example, a new queue priority rule can be programmed, given a name, and then used within the specification language. However, this is usually more difficult than extending a simulation language to do the same thing.

14.5.3 Natural language interfaces

As *humans* can describe a system to be simulated in English, it seems reasonable to investigate methods that allow this description to be accepted by computer. Doukidis and Paul (1985) have produced an experimental system which allows models to be expressed in a constrained form of English. For example, given

THE CAT SAT ON THE MAT

their system assumes that CAT is an entity, which requires a MAT for the activity of SITTING. A dialog is then pursued by the system, which will aim to construct activity-cycles for both CAT and MAT.

The system can produce a skeletal model, which can then be extended as desired. It is unlikely that natural language development interfaces can ever do much more than this (at least general purpose ones). The need for a specific language in simulation is like the need for mathematics; English, or any other natural language, is too general to capture the necessary level of detail.

14.6 DOMAIN DEPENDENT TOOLS

Tools developed for specific domains (e.g., manufacturing, health care,

traffic flow, etc.) can overcome many of the disadvantages of general purpose simulation tools, since:

(*a*) the range of model types that need to be dealt with is limited,
(*b*) the technical language of that domain can be used within the tool, rather than an abstract language (e.g., 'machines' or 'ships' rather than 'entities'),
(*c*) methods of analysis applicable to the particular domain can be embodied in the tool.

Thus it is not surprising that a number of domain-dependent development tools exist. Many of these have been developed for the simulation of manufacturing systems. For example, see the work by Haddock (1987) on a program generator for SIMAN. His system also executes the model, and automatically performs some steady-state analyses. Readers interested in developing domain-dependent tools, should consider using a package based on a high-level language such as Pascal_SIM because the code is available to be adapted for specific purposes.

14.7 MODEL DEVELOPMENT ENVIRONMENTS

Chapter 13 showed that a simulation program must be regarded as just one part of a computer system for decision-making. A system must be analysed, data collected, and distributions fitted to the data; the program must be verified; the model must be validated; experiments must be run, and the results analysed and displayed. These parts of the simulation process, which are as important as programming the model logic, also need software support.

Sometimes, readily available software can be employed. Chapter 13 recommended the use of spreadsheets and statistical packages and considered the advantages of using a database with a simulation.

However, there are a number of advantages to using a single environment for all of the simulation process. All data can reside in one format in one place, and the developer can use a single set of commands across all of the simulation tasks. Thus there are a number of attempts to construct general purpose simulation development environments which provide tools for the entire process of simulation modelling and development. These could include:

(*a*) programming tools, such as a simulation programming language;
(*b*) data analysis tools, including a distribution fitting tool;
(*c*) data management tools, allowing for the analysis and use of both collected data and data generated by the simulation;

(*d*) model management tools, providing facilities for saving and reusing all or part of previously developed models;
(*e*) documentation tools, which aid the easy production of complete documentation;
(*f*) animation tools, for the design, management, and reuse (in other simulation models) of pictures.

For a complete discussion of the concept, see Balci (1986).

The only commercially available environment is TESS (Standridge 1985), which is based upon the SLAM II simulation language. TESS provides facilities for data management, model development (which can be done through a network diagram interface), and animation. It is likely that a number of environments will appear in the future, hosted on high quality work stations. Like TESS, many will be based upon existing simulation languages.

14.8 SUMMARY

Advances in computing and simulation have produced significant developments for the simulation developer and user:

(*a*) visual output can be designed and produced using an animator;
(*b*) complex decision mechanisms can be modelled with knowledge representation techniques;
(*c*) prescriptive simulation techniques can free the user from the need to design, run, and analyse experiments;
(*d*) advanced development tools make the development of some simulations easier and faster;
(*e*) packages are available for specific application areas;
(*f*) model development environments give the user a complete set of tools to support the entire simulation process.

At their present stage of development, some of these advances have certain drawbacks. New ideas are being evolved which link knowledge representation with simulation, prescriptive simulation, and model development environments. Present implementations tend to be both restricted in scope, and inefficient. The present range of animators and advanced development tools are only appropriate for a limited set of situations and modelling approaches. Moreover, for the most part, these developments have been made independently of each other, using different hardware and software.

A comprehensive package of facilities incorporating most of the above could be developed in Pascal. The resulting system could be run on

many different computers. The major problem is that Pascal is unsuitable for knowledge representation work. However, it would be possible, and might be more efficient, to query Prolog from a Pascal simulation, rather than construct the entire simulation in Prolog.

Whatever advances take place in simulation, the discrete structures taught in this book are fundamental to all discrete-event simulation programming work. Pascal_SIM is simple, fast, and efficient, and may be used as it stands, or built upon to provide more complex facilities. Alternatively the theory and structures taught here may be translated to another package, or may be used to develop simulation facilities in a different programming language.

Appendix A

Pascal_SIM Documentation

A.1 INTRODUCTION

A.1.1 ISO Pascal

For the most part, the Pascal facilities provided in the package called Pascal_SIM, follow the ISO standard for Pascal (Wilson and Addyman 1982). These are three areas where this standard is not adhered to:

(*a*) the use of a *string* type,
(*b*) the use of built-in string manipulation functions *concat* and *length* in the procedure **print_histogram**,
(*c*) the use of the underscore character in variable names.

However, most implementations provide a string type with associated routines, and (if necessary) underscores can be easily edited out. Implementation details, relating to specific systems, are covered in Appendix C.

A.1.2 Multiple copies

Note that amongst the code appearing in Appendix B there are two copies of:

(*a*) *function* **rnd** (*s* :**stream_num**) :*real* – one for 16-bit integer machines, one for 32-bit integer machines.
(*b*) *procedure* **set_foreground** (*c* :**color**) and *procedure* **set_background** (*c* :**color**) – one copy of each implements user-installed screen control codes, presently set for extended ANSI, and one copy of each translates into calls to Turbo Pascal built-in routines.
(*c*) *procedure* **run** (*duration* :*real*; *max c* :**cardinal**) – one for programs that use the three-phase and event approaches, and one for programs using the process description approach.

A.1.3 Structure of Pascal_SIM program

A three-phase simulation program using Pascal_SIM has the following structure:

```
{ B events }
procedure B1;
procedure B2;
  :
  :
{ C events }
procedure C1;
procedure C2;
  :
  :
procedure display;
procedure run (duration :real; max_C :cardinal);
procedure initialize;
procedure picture;
procedure report;
  begin
    :
  initialize;
  picture;
    :
  run ( ..., ...);
    :
  report;
    :
  end.
```

Where: .

(*a*) procedure **display** is called after every advance of the clock, and contains details of visual display changes necessary after every clock advance,
(*b*) procedures **initialize** and **picture** should each be called just once at the start of a simulation to initialize the simulation data structures and the static visual display respectively,
(*c*) procedure **report** should be called after completion of the simulation to produce the statistics,
(*d*) the user needs to enter the exact titles and parameter lists of each bound event and conditional event in the executive.

If a pure event-based approach is used, the structure is the same except that no conditional events will be present. For process description models, a procedure will be programmed for each process rather than each event, and the process version of the executive must be used.

A.1.4 Executive mechanisms

Here, the time flow mechanisms used by the two executives are presented. The main executive,

procedure **run** (*duration* :*real*; *max_c* :**cardinal**);

uses the three-phase approach thus:

A: < advance time to the time of the next bound event >;
B: < for any bound events due to happen at this time >
< take the entity off the calendar >;
< execute the event >;
C: < attempt all conditional events >;

The process interaction executive,

procedure **run** (*duration* :*real*);

is based upon the executive found in GPSS. Here an entity is always either on the next event calendar or on a chain of suspended processes. The conditional event scan is replaced by a scan of the chain of suspended processes:

```
A: < advance time to the time of the next event >;
B: < for any events due to happen at this time >
     < take the entity off the calendar >;
     < reactivate the entity's process >;
     if < it has finished its process >
       then < dispose of the entity >
       else
         if < the entity is not on the calendar > then
           < put it on the end of the chain of suspended processes >;
C: repeat
     for < each entity on the suspended chain >
       < reactivate it >;
       if < it has advanced > or < is on the calendar >
         then < remove it from the suspended chain >;
              < dispose of the entity if it has finished its process >
         else
           if < it is not on the calendar > and
             < it has advanced >
             then < put it on the end of the chain
                    of suspended processes >;
   until < the chain has been scanned >
     and < no processes have been activated >;
```

Note that each executive will cycle through the two or three phases until either:

(*a*) the next B event is beyond the duration of the simulation run, or
(*b*) the calendar is empty.

A.1.5 Error messages

A number of error messages can be produced by the Pascal_SIM routines. Most are concerned with detecting undesirable parameters, (such as 'nil' pointers) when the simulation is running. In each case the simulation is brought to an orderly halt. The possible messages, and their causes, are as follows.

(*a*) *Adding a non-existent entity* (parameter *i* is *nil* in **give**).
(*b*) *Taking a nil link* (parameter *l* is *nil* in **take**).
(*c*) *Taking a queue head* (parameter *t* is the queue head in **take**).
(*d*) *Disposing of a busy entity* (parameter *e* points to an entity that is not available in **dis_entity**).
(*e*) *Causing a non-existent entity* (parameter *e* is *nil* in **cause**).
(*f*) *Causing an entity already entered in the calendar.*
(*g*) *Causing time negative* (parameter *t* is negative in **cause**).
(*h*) *Acquiring from bin* (a request for more resources than presently available in **acquire**).
(*i*) *Returning to bin* (an attempt to return more resources than are out in **return**).
(*j*) *Sample file finished* (the end of the sample has been reached prior to the proper creation of a **lookup_table** in **make_sample**).
(*k*) *Size of cell in sample* (**make_sample** has encountered a cell in the provided sample file with a value less than 0·0 or greater than 1·0).
(*l*) *Order of sample file* (cell values are not in ascending order in **make_sample**).
(*m*) *Histogram is empty* (an attempt to print an empty histogram in **print_histogram**).
(*n*) *Moving vertical entity* (first *y* ordinate *y1* is less than or equal to second ordinate *y2* in **move_v**).
(*o*) *Moving horizontal entity* (first *x* ordinate *x1* is greater than or equal to second ordinate *x2* in **move_h**).

In addition, most Pascal systems have range checking facilities which automatically check, when the program is running, whether scalar values are within the range specified by the programmer. Note, however, that a number of compilers have the range checking turned off as the default option.

A.2 DOCUMENTATION

Every constant, type, global variable, function and procedure is documented below. The information has the following format.

(*a*) *Meaning* – a brief description. Where appropriate, individual fields of records and parameters are described.
(*b*) *Uses* (functions and procedures only) – any global variables used.
(*c*) *Errors* (functions and procedures only) – any error messages that may be produced are presented in the form description (actual message).
(*d*) *Calls* (functions and procedures only) – any functions or procedures called.
(*e*) *Used by* – any type or variable that utilizes the item being described.
(*f*) *Used in* – any function or procedure that utilizes the item being described.

A.2.1 Constants

max_cell_num = 16;
meaning : the maximum number of cells in a histogram
used by : **cell_num**
used in : **reset_histogram, print_histogram, log_histogram**
max_stream_num = 32;
meaning : the maximum number of streams
used by : **stream_num**
used in : **make_streams**
max_class_num = 256;
meaning : the maximum number of classes in the class table
used by : **class_num**
max_sample_num = 20;
meaning : the maximum number of samples in a lookup table
used in : **make_sample**
max_string_length = 80;
meaning : the maximum number of characters in a string
used by : **a_string**
delay_num = 2000;
meaning : the delay of the visual display
used in : **delay**

A.2.2 Types

a_string = **string [max_string_length]**;
meaning : a string of **max_string_length** characters
used in : **sim_error, print_histogram**
cardinal = 0..**maxint**;
meaning : non-negative integer
used by : **an_entity, bin, original_seeds, seeds**
used in : **count, make_class, branch, make_streams, poisson, print_histogram, log_histogram, delay, make_class_table, write_block, write_time**
color = **(nul,black,red,green,yellow,blue,magenta,cyan,white)**;
meaning : colors used for both foreground and background in visual displays
used by : **an_entity, class_table**
used in : **set_foreground, set_background, write_queue, enter_class, write_block, move_v, move_h**
stream_num = 1..**max_stream_num**;
meaning : random number stream
used by : **original_seeds, seeds**
used in : **rnd, make_streams, normal, log_normal, poisson, negexp, uniform**
cell_num = 0..**max_cell_num**;
meaning : histogram cell, where cell number 0 is for underflow and cell

number **max_cell_num** is for overflow
used by : **histogram**
used in : **reset_histogram, print_histogram**

class_num = 1..**max_class_num**;
meaning : number of classes in the class table
used by : **an_entity, class_table**
used in : **make_class_table, enter_class**

sample_num = 1..**max_sample_num**;
meaning : number of samples in a lookup table
used by : **lookup_table**
used in : **make_sample, sample**

string_length = 1..**max_string_length**
meaning : permissible number of characters in a string
used in : **print_histogram**

entity = ^**an_entity**;
meaning : pointer to an entity
used by : **a_link, current**
used in : **give, take, give_top, give_tail, take_top, take_tail, new_entity, dis_entity, make_class, calendar_top, write_entity, move_v, move_h**

link = ^**a_link**;
meaning : pointer to a link
used by : **a_link**
used in : **give, take, count, make_class, cause, write_queue**

a_link = *record*
next,pre :**link**;
item :**entity**;
end;
meaning : links from which queues are built
next – the next link in the queue
pre – the previous link in the queue
item – the entity connected to the link
used by : **link**
used in : (fields of record)
make_queue, give, take, count, make_class, cause

queue = **link**;
meaning : a queue (although identical to a **link**, used where an entire queue is implied)
used by : **calendar, suspended_chain**
used in : **make_queue, give, take, give_top, give_tail, take_top, take_tail**

an_entity = *packed record*
avail :*boolean*;
class :**class_num**;
col :**color**;
attr,next_b :**cardinal**;
time :*real*;
end;

meaning : the record for an entity
avail – false if the entity is entered in the calendar
class – where the entity is entered in the **class_table**, if the class table is used
col – the entity's present color
attr – the entity's individual attribute number
next_b – the next bound event or block that the entity will enter
used by : **entity**
used in : **new_entity, dis_entity, count, calendar_top, branch, remove_entity, write_entity**

bin = *record*
number,num_avail :**cardinal**;
end;
meaning : a bin of resources
number – the total number of resources in the bin
num_avail – the number of resources presently available
used in : **make_bin, acquire, return**

histogram = *packed record*
cell :*array* [**cell_num**] *of real*;
count,width,base,
total,sosq,
min,max,expended :*real*;
state :*boolean*;
end;
meaning : a histogram used for statistics collection
cell – the histogram cells
count – the number of observations
width – the cell width
base – the cell base
total – the total of all observations
sosq – sum of square of all observations
min – the minimum observation
max – the maximum observation
expended – the last time at which an observation was logged
state – true if a state histogram, false if a time weighted histogram.
used in : **reset_histogram, make_histogram, print_histogram, log_histogram**

lookup_table = *array* [*1*..**max_sample_num**,1..2] *of real*;
meaning : an empirical distribution
used in: **make_sample, sample**

A.2.3 Global variables

tim :*real*;
meaning : the present clock time
used in : **run, cause, write_time**
current :**entity**;
meaning : the entity that has caused the most recent bound event or whose process is active
used in : **calendar_top, cause, run**
calendar :**queue**;
meaning : the calendar of future events (sometimes known as the next event set)
used in : **make_sim, calendar_top, cause, run**
on_calendar :*boolean*;
meaning : true if the current entity has been placed on the calendar
used in : **cause, run** (process version only)
suspended_chain :**queue**;
meaning : the chain of suspended entities in a process description model
used in : **run** (process version only)
running :*boolean*;
meaning : true if the simulation is running; can be set to false to bring the simulation to an orderly halt
used in : **sim_error,run**
original_seeds :*array* [**stream_num**] *of* **cardinal**;
meaning : the original seeds used by the random number generator
used in : **make_streams**
seeds :*array* [**stream_num**] *of* **cardinal**;
meaning : the present values of the seeds, as updated by the random number generator
used in : **make_streams, rnd**
class_table :*array* [**class_num**] *of*
record
let :**char**; **col** :**color**;
end;
meaning : the table of letters and base colors used to display entities from a class in the visual display
let – letter
col – color
used in : **make_class_table, enter_class, write_entity**

A.2.4 FUNCTIONS AND PROCEDURES

A.2.4.1 Queue processing

procedure **make_queue** (*var q* :**queue**);
meaning : make a new queue
q – a previously declared queue
calls : **new_entity**
used in : **make_class, make_sim**
procedure **give** (*q* :**queue**; *t* :**link**; *i* :**entity**);
meaning : give an entity to a queue
q – a previously declared queue
t – a link in *q*; new link is placed immediately after *t*
i – an active entity
errors : *i=nil* (giving a *nil* entity)
used in : **give_top, give_tail, cause**
calls : **sim_error**
function **take** (*q* :**queue**; *t* :**link**) :**entity**;
meaning : remove a link from a queue, and return the connected entity
q – a previously declared queue
errors : *t* is *nil* (taking a *nil* link), *t=q* (taking a queue head)
used in : **take_top, take_tail, calendar_top**
calls : **sim_error**
procedure **give_top** (*q* :**queue**; *i* :**entity**);
meaning : give an entity to the top of a queue
q – a previously declared queue
i – an active entity
calls : **give**
procedure **give_tail** (*q* :**queue**; *i* :**entity**);
meaning : give an entity to the tail of a queue
q – a previously declared queue
i – an active entity
used in : **run** (process version only)
calls : **give**
function **take_top** (*q* :**queue**) :**entity**;
meaning : remove the link at the top of a queue, and return the connected entity
q – a previously declared queue
used in : **calendar_top**
calls : **take**
function **take_tail** (*q* :**queue**) :**entity**;
meaning : remove the link from the tail of a queue, and return the connected entity
q – previously declared queue
calls : **take**
function **empty** (*q* :**queue**) :**boolean**;
meaning : returns true if queue *q* is empty

A.2.4.2 Entities and classes

function **new_entity** (*c* :**class_num**; *a* :**attr_num**) :**entity**;
meaning : create and return a new entity record
c – required **class_table** entry
a – required attribute number
used in : **make_class**, **make_queue**
procedure **dis_entity** (*e* :**entity**);
meaning : dispose of an entity
e – entity to be disposed of
errors : *e* is entered in the calendar (disposing of a busy entity)
procedure **make_class** (*var c* :**queue**; *n,size* :**cardinal**);
meaning : make a class of entities
c – queue to hold class
n – number of entry in **class_table**
size – number of entities in the class;
attribute numbers are assigned from 1 to *n*
calls : **make_queue**, **new_entity**
function **count** (*var q* :**queue**) :**cardinal**;
meaning : return the number of entities on a queue
q – a previously declared queue

A.2.4.3 Timing and executive

procedure **make_sim**;
meaning : initialize the simulation data structures;
create the calendar, set **tim** to 0
uses : **calendar**, **tim**
procedure **cause** (*nb* :**cardinal**; *e* :**entity**; *t* :*real*);
meaning : cause an event or entry to a block to happen
nb – number of the event or block
e – the entity that is entered in the calendar
t – the event occurs at time **tim**+*t*
on_calendar is set to true if *e* is the current entity
uses : **calendar**, **tim**, **on_calendar**, **current**
errors : *e* is a *nil* pointer (causing a non-existent entity)
t is less than zero (causing time negative)
calls : **give**
procedure **calendar_top**;
meaning : remove the entity that enters the next event from the calendar;
this entity becomes the **current** entity and is set to available
used in : **run** (both versions)

A.2.4.4 Facilities for process executive

procedure **branch** (*next* :**cardinal**);
meaning : set the next block that the current entity should immediately attempt
next – the number of the next block
uses : **current**
procedure **remove_entity**;
meaning : mark the current entity for removal from the simulation
uses : **current**

A.2.4.5 Resources

procedure **make_bin** (*var from* :**bin**; *n* :**cardinal**);
meaning : create a bin of resources
from – a previously declared bin
n – the number of resources in the bin, all of which are initially available
procedure **acquire** (*var from* :**bin**; *n* :**cardinal**);
meaning : acquire a number of resources from a bin
from – a previously declared bin
n – the number to be acquired
errors : less than *n* resources are presently available (acquiring from bin)
procedure **return** (*var from* :**bin**; *n* :**cardinal**);
meaning : return a number of resources to a bin
from – a previously declared bin
n – number to be returned
errors : less than *n* resources are presently in use (returning to bin)

A.2.4.6 Error messages

procedure **sim_error** (*s* :**a_string**);
meaning : halt the simulation (by setting **running** to false)
s – used in error message
uses : **running**

A.2.4.7 Random number generator and streams

procedure **make_streams**;
meaning : create the random number streams; streams numbered 1 to **max_stream_num** are assigned the seeds 1007, 2007, 3007 etc.
uses : **original_seeds**, **seeds**
function **rnd** (*s* :**stream_num**) :*real*;
meaning : returns a continuous uniform pseudo-random number in the closed range 0·0 to 1·0; two versions are present
uses : **seeds**
used in : **normal**, **poisson**, **negexp**, **uniform**, **sample**

A.2.4.8 Sampling distributions

function **normal** (*m,sd* :*real*; *s* :**stream_num**) :*real*;
meaning : returns a normal variate
m – mean of normal distribution
sd – standard deviation of the distribution
s – random number stream
calls : **rnd**

function **log_normal** (*m,sd* :*real*; *s* :**stream_num**) :*real*;
meaning : returns a log normal variate
m – mean of log normal distribution
sd – standard deviation of the distribution
s – random number stream
calls : **normal**

function **poisson** (*m* :**real**; *s* :**stream_num**) :**cardinal**;
meaning : returns a Poisson variate
m – mean of Poisson distribution
s – random number stream
calls : **rnd**

function **negexp** (*m* :**real**; *s* :**stream_num**) :*real*;
meaning : returns a negative exponential variate
m – mean of negative exponential distribution
s – random number stream
calls : **rnd**

function **uniform** (*l,h* :*real*; *s* :**stream_num**) :*real*;
meaning : returns a continuous uniform variate
l – lower bound value
h – higher bound value
s – random number stream
calls : **rnd**

procedure **make_sample** (*var sample_file* :*text*; *var table* :**lookup_table**);
meaning : creates a discrete lookup table
sample_file – a previously opened file containing the values
table – a previously declared lookup table
errors : (a) the end of file marker is encountered before the lookup table is complete (sample file finished)
(b) a cell value has been found that is less than 0·0 or greater than 1·0 (size of cell in sample)
(c) a cell value has been found that is smaller than the previous cell (order of sample file)

function **sample** (*table* :**lookup_table**; *s* :**stream_num**) :*real*;
meaning : return a sample from a lookup table
table – a previously declared and created lookup table
s – random number stream
calls : **rnd**

A.2.4.9 Histograms

procedure **reset_histogram** (*var h* :**histogram**);
meaning : reset a histogram; all cells and accumulators are set to zero, **expended** is set to the present clock time **tim**

procedure **make_histogram** (*var h* :**histogram**; *cell_base*, *cell_width* :*real*; *s*:*boolean*);
meaning : create a new histogram
h – a previously declared histogram
cell_base – the base value of first cell
cell_width – width of each cell
state – set to S
calls : **reset_histogram**

procedure **print_histogram** (*var pr* :*text*; *h* :**histogram**; *len* :**string_length**);
meaning : print a histogram
pr – a previously opened text file
h – the histogram to be printed
len – the maximum number of asterisks in a cell
errors : *h* **count** is 0 (histogram is empty)

procedure **log_histogram** (*var h* :**histogram**; *x,y* :*real*);
meaning : log an observation to a histogram
h – the histogram
x – identifies the cell
y – identifies what is added
(for a state histogram, *y* is added to the appropriate cell for a time weighted histogram, **tim-expended** is added, and **expended** is set to the new clock time **tim**)

A.2.4.10 Screen control

procedure **make_screen**;
meaning : initialize the console

procedure **gotoxy** (*x,y* :**cardinal**);
meaning : move the console cursor to co-ordinates (*x,y*), where (1,1) is the top left to the screen
used in : **write_entity**, **write_queue**, **write_block**, **move_v**, **move_h**

procedure **clear_screen**;
meaning : clear the screen

procedure **set_foreground** (*c* :**color**);
meaning : set the foreground text color
c – the color to be used
(a second version translates calls to **set_foreground** into calls to the built in Turbo Pascal routine *textcolor*)
used in : **write_entity**

procedure **set_background** (*c* :**color**);
meaning : set the background text color
c – the color to be used
(a second version translates calls to **set_background** into calls to the built in Turbo Pascal routine *textbackground*)
used in : **write_queue**, **write_block**, **move_v**, **move_h**

procedure **reset_colors**;
meaning : set the foreground and background colors to the default state (white foreground, black background)
used in : **write_queue**, **write_block**, **move_v**, **move_h**

A.2.4.11 Visual displays

procedure **delay**;
meaning : delay the whole program, and hence the display, for **delay_num** empty **for** loops
used in : **move_v**, **move_h**
procedure **make_class_table**;
meaning : initialize the **class_table**
uses : **class_table**
procedure **enter_class** (*n* :**class_num**; *l* :*char*; *c*:**color**);
meaning : enter a class in the class table
n – the number of the class
l – letter to be used to represent entities from a class on the visual display
c – color for letters representing entities, unless an entity's own **col** field has been set
uses : **class_table**
procedure **write_entity** (*x*,*y* :**cardinal**; *e* :**entity**);
meaning : display an entity on the console
x,*y* – co-ordinate pair
e – the entity
uses : **class_table**
calls : **gotoxy**, **set_foreground**
used in : **write_queue**, **move_v**, **move_h**
procedure **write_queue** (*x*,*y* :**cardinal**; *b* :**color**; *q* :**queue**; **max_length** :**cardinal**);
meaning : display a queue on the console
x,*y* – co-ordinate pair
b – background color for the queue
q – the queue
max_length – the maximum length of display
calls : **set_background**, **write_entity**, **gotoxy**, **reset_colors**
procedure **write_block** (*x1*,*y1*,*x2*,*y2* :**cardinal**; *b* :**color**);
meaning : display a background block of color
x1,*y1* – co-ordinate pair
x2,*y2* – co-ordinate pair, where x1<x2 and y1<y2
b – background color
calls : **set_background**, **gotoxy**, **reset_colors**

procedure **move_v** (*x*,*y1*,*y2* :**cardinal**; *e* :**entity**; *b* :**color**);
meaning : move an entity in the display vertically
x – vertical file
y1 – from
y2 – to, where *y1*<*y2*
b – background color
errors : *y1*>=*y2* (moving vertical entity)
calls : **set_background**, **write_entity**, **gotoxy**
procedure **move_h** (*y*,*x1*,*x2* :**cardinal**; *e* :**entity**; *b* :**color**);
meaning : move an entity in the display horizontally
y – horizontal file
x1 – from
x2 – to, where *x1*<*x2*
b – background color
errors : *x1*>=*x2* (moving horizontal entity)
calls : **set_background**, **write_entity**, **gotoxy**
procedure **write_time**;
meaning : write the present clock time in the form day: hour: minute

A.2.4.12 User-written routines

These are all routines that are to be provided by the user.

procedure **display**;
meaning : visual display alterations performed prior to advancing the time to the next event
procedure **initialize**;
meaning : initialize the simulation
calls : **make_sim**, **make_streams**
procedure **picture**;
meaning : initialize the static part of the visual display
calls : **make_class_table**
procedure **report**;
meaning : final report and statistics from a simulation run

A.2.4.13 Simulation executive

procedure **run** (*duration* :*real*; *max_c* :**cardinal**);
(or for the process description executive,
procedure **run** (*duration* :*real*;)
meaning : run the simulation until **duration** is reached or **finished** has been set to **true**
duration – the maximum duration of run
max_c – the number of C events
uses : **current**, **calendar**, **tim**, **running**
on_calendar (process version only)
calls : **calendar_top**

Appendix B

Pascal_SIM

```
const
  max_cell_num      = 16;
  max_stream_num    = 32;
  max_class_num     = 256;
  max_sample_num    = 20;
  max_string_length = 80;
  delay_num         = 2000;

type
  a_string = string [max_string_length];
  cardinal = 0..maxint;
  color =  (nul,black,red,green,yellow,blue,magenta,cyan,
            white);
  stream_num    = 1..max_stream_num;
  cell_num      = 0..max_cell_num;
  class_num     = 1..max_class_num;
  sample_num    = 1..max_sample_num;
  string_length = 1..max_string_length;
  entity = ^an_entity;
  link   = ^a_link;
  a_link = record
             next,pre :link;
             item     :entity;
           end;
  queue  = link;
  an_entity = packed record
                avail       :boolean;
                class       :class_num;
                col         :color;
                attr,next_B :cardinal;
                time        :real;
              end;
  bin = record
          number,num_avail :cardinal;
        end;
  histogram = record
                cell                :array [cell_num] of real;
                count,width,base,
                  total,sosq,
                  min,max,expended :real;
                state               :boolean;
              end;
```



```
  lookup_table = array [sample_num,1..2] of real;

var
  tim                 :real;
  current             :entity;
  calendar            :queue;
  on_calendar         :boolean;
  suspended_chain     :queue;
  running             :boolean;
  original_seeds,seeds :array [stream_num] of cardinal;
  class_table         :array [class_num] of
                         record
                           let :char; col :color;
                         end;

{ available routines are

- error messages -
procedure sim_error (s :a_string);

- queue processing -
procedure make_queue (var q :queue);
procedure give (q :queue; t :link; i :entity);
function  take (q :queue; t :link) :entity;
procedure give_top (q :queue; i :entity);
procedure give_tail (q :queue; i :entity);
function  take_top (q :queue) :entity;
function  take_tail (q :queue) :entity;
function  empty (q :queue) :boolean;

- entities and classes -
function  new_entity (c :class_num; a :cardinal) :entity;
procedure dis_entity (e :entity);
function  count (q :queue) :cardinal;
procedure make_class (var c :queue; n,size :cardinal);

- timing and the executive -
procedure calendar_top;
procedure make_sim;
procedure cause (nb :cardinal; e :entity; t :real);

- facilities for process executive -
procedure branch (next :cardinal);
procedure remove_entity;

- resources -
procedure make_bin (var from :bin; n :cardinal);
procedure acquire (var from :bin; n :cardinal);
procedure return (var from :bin; n :cardinal);

- random number generation and streams -
procedure make_streams;
function rnd (s :stream_num) :real;
```

```
  - distribution sampling -
  function normal (m,sd :real; s :stream_num) :real;
  function log_normal (m,sd :real; s :stream_num) :real;
  function poisson (m :real; s :stream_num) :cardinal;
  function negexp (m :real; s :stream_num) :real;
  function uniform (l,h :real; s :stream_num) :real;
  procedure make_sample (var sample_file :text;
                         var table :lookup_table);
  function sample (table :lookup_table; s :stream_num) :real;

  - histograms -
  procedure reset_histogram (var h :histogram);
  procedure make_histogram (var h :histogram;
                    cell_base,cell_width :real; s :boolean);
  procedure print_histogram (var pr:text;h:histogram;
                                     len :string_length);
  procedure log_histogram (var h :histogram; x,y :real);

  - screen control -
  procedure make_screen;
  procedure gotoxy (x,y :cardinal);
  procedure clear_screen;
  procedure set_foreground (c :color);
  procedure set_background (c :color);
  procedure reset_colors;

  - visual display -
  procedure delay;
  procedure make_class_table;
  procedure enter_class (n :class_num; q :char; c :color);
  procedure write_entity (x,y :cardinal; e :entity);
  procedure write_queue (x,y :cardinal; b :color; q :queue;
                                    max_length :cardinal);
  procedure write_block (x1,y1,x2,y2 :cardinal; b :color);
  procedure move_v (x,y1,y2 :cardinal; e :entity; b :color);
  procedure move_h (y,x1,x2 :cardinal; e :entity; b :color);
  procedure write_time;

  - dummy procedures -
  procedure display;
  procedure initialize;
  procedure picture;
  procedure report;

  - simulation executive -
  procedure run (duration :real; max_C :cardinal); }

  { error messages }
  procedure sim_error (s :a_string);
    begin
      writeln ('****  ERROR  ',s,'   ***'); running := false;
    end { simulation error };
```



```
  { queue processing }
  procedure make_queue (var q :queue);
  { note :'new_entity must be forward declared }
    begin
      new (q);
      with q^ do
        begin
          item := new_entity (1,0);
          next := q; pre := q;
        end;
    end { make queue };

  procedure give (q :queue; t :link; i :entity);
    var
      new_pt :link;
    begin
      if i = nil then sim_error('adding a non-existent entity')
        else begin
               new (new_pt);
               with t^ do
                 begin
                   new_pt^.next := next; next^.pre := new_pt;
                   next := new_pt; new_pt^.pre := t;
                 end;
               new_pt^.item := i;
             end;
    end { give };

  function take (q :queue; t :link) :entity;
    begin
      if t = nil then sim_error ('taking a nil link')
        else
          if t = q then sim_error ('taking a queue head')
            else begin
                   with t^ do
                     begin
                       pre^.next := next; next^.pre := pre;
                     end;
                   take := t^.item;
                   dispose (t);
                 end;
    end { take };

  procedure give_top (q :queue; i :entity);
    begin
      give (q,q^,i);
    end { give top };

  procedure give_tail (q :queue; i :entity);
    begin
      give (q,q^.pre,i);
    end { give tail };
```

```
function  take_top (q :queue) :entity;
  begin
    take_top := take (q,q^.next);
  end { take top };

function  take_tail (q :queue) :entity;
  begin
    take_tail := take (q,q^.pre);
  end { take tail };

function empty (q :queue) :boolean;
  begin
    empty := (q = q^.next);
  end { empty };

{ entities and classes }
function new_entity (c :class_num; a :cardinal) :entity;
  var
    e :entity;
  begin
    new (e);
    with e^ do
      begin
        time := 0.0; avail := true; class := c;
        col := nul; attr := a; next_B := 0;
      end;
    new_entity := e;
  end { new entity };

procedure dis_entity (e :entity);
  begin
    if not e^.avail then
      sim_error ('disposing of a busy entity')
      else dispose (e);
  end { dispose entity };

function count (q :queue) :cardinal;
  var
    total :cardinal;
    t     :link;
  begin
    total := 0; t := q^.next;
    while q<>t do
      begin
        total := total+1; t := t^.next;
      end;
    count := total;
  end { count };
```



```
procedure make_class (var c :queue; n,size :cardinal);
  var
    t :link;
    i :cardinal;
    e :entity;
  begin
    make_queue (c);
    t := c;
    for i := 1 to size do
      begin
        e := new_entity (n,i);
        give (c,t,e);
        t := t^.next;
      end;
  end { make class };

{ timing and the executive }
procedure calendar_top;
  var
    e :entity;
  begin
    e := take (calendar,calendar^.next);
    e^.avail := true;
    current := e;
  end { calendar top };

procedure make_sim;
  begin
    make_queue (calendar);
    make_queue (suspended_chain);
    calendar^.item := new_entity (1,0);
    tim := 0.0;
  end { make sim };

procedure cause (nb :cardinal; e :entity; t :real);
  var
    l       :link;
    new_tim :real;
  begin
    if e = nil then
      sim_error ('causing a non-existent entity')
    else
      if not e^.avail then
        sim_error
       ('causing an entity already entered in the calendar')
        else
          if t<0.0 then sim_error ('causing time negative')
            else begin
                   new_tim := tim+t;
                   with e^ do
                     begin
                       time := new_tim; next_B := nb;
                       avail := false;
                     end;
```

```
                    l := calendar^.next;
                    while (new_tim>l^.item^.time)
                    and (l<>calendar)
                      do l := l^.next;
                    l := l^.pre;
                    give (calendar,l,e);
                    if e=current then on_calendar := true;
                  end;
  end { cause };

{ facilities for process executive }
procedure branch (next :cardinal);
  begin
    with current^ do next_B := next;
  end { branch };

procedure remove_entity;
  begin
    current^.next_B := 0;
  end { remove entity };

{ resources }
procedure make_bin (var from :bin; n :cardinal);
  begin
    with from do
      begin
        number := n; num_avail := n;
      end;
  end { make bin };

procedure acquire (var from :bin; n :cardinal);
  begin
    with from do
      if n>num_avail then sim_error ('acquiring from bin')
        else num_avail := num_avail-n;
  end { acquire };

procedure return (var from :bin; n :cardinal);
  begin
  with from do
    if (n+num_avail>number) or (n<0) then
      sim_error ('returning to bin')
      else num_avail := num_avail+n;
  end { return };
```



```
{ random number generation and streams }
function rnd (s :stream_num) :real;
{ for 16 bit integers - assumes no detection of overflow }
  const
    mult  = 3993;
    divid = maxint;
    add   = 1;
  begin
    seeds[s] := (seeds[s]*mult+add) mod divid;
    if seeds[s]<0 then seeds[s] := seeds[s]+maxint+1;
    rnd := seeds[s]/divid;
  end { random number generator };

function rnd (s :stream_num) :real;
{ generator for 32 bit integers }
  const
    mult  = 16807.0;
    divid = 2147483647.0;
  var
    r,quotient :real;
  begin
    r := mult*seeds[s];
    quotient := r/divid;
    seeds[s] := trunc (r-(trunc(quotient)*divid));
    rnd := seeds[s]/divid;
  end { random number generator };

procedure make_streams;
  var
    i : stream_num;
  begin
    for i := 1 to max_stream_num do
      original_seeds[i] := i*1000+7;
    seeds := original_seeds;
  end { make streams };

{ distribution sampling }
function normal (m,sd :real; s :stream_num) :real;
  const
    tpi = 6.2831852;
  begin
    normal := sin (tpi*rnd(s))*sqrt(-2*ln (rnd (s)))*sd+m;
  end { normal };

function log_normal (m,sd :real; s :stream_num) :real;
  var
    x,xs,xm :real;
  begin
    x := ln (sd/m+1);
    xs := sqrt (x);
    xm := ln (m)-0.5*x;
    log_normal := exp (normal(xm,xs,s));
  end { log normal };
```

```
function poisson (m :real; s :stream_num) :cardinal;
  var
    p,f,u :real;
        x :cardinal;
  begin
    p := exp (-m);
    f := p; x := 0;
    u := rnd (s);
    while u>f do
      begin
        x := x+1;
        p := (m/x)*p;
        f := f+p;
      end;
    poisson := x;
  end { poisson };

function negexp (m :real; s :stream_num) :real;
  begin
    negexp := -ln (rnd(s))*m;
  end { negexp };

function uniform (l,h :real; s :stream_num) :real;
  begin
    uniform := l+(h-l)*rnd (s);
  end { uniform };

procedure make_sample
          (var sample_file :text; var table :lookup_table);
  var
    i        :sample_num;
    last     :real;
    finished :boolean;
  begin
    for i := 1 to max_sample_num do
      begin
        table[i,1] := 0; table[i,2] := 0;
      end;
    i := 1; last := 0;
    finished := false;
    repeat
      if eof (sample_file) then
        begin
          finished := true;
          sim_error ('sample file finished');
        end
      else
        begin
          readln (sample_file,table[i,1],table[i,2]);
          if (table[i,1]<0.0) or (table[i,1]>1.0) then
            begin
              finished := true;
              sim_error ('size of cell in sample');
            end;
```



```
          if table[i,1] <= last then
            begin
              finished := true;
              sim_error ('order of sample file');
            end
          else
            begin
              last := table[i,1];
              i := i+1;
              if last = 1.0 then finished := true;
            end;
        end;
    until finished;
  end { make sample };

function sample (table :lookup_table; s :stream_num) :real;
  var
    i :sample_num;
    u :real;
  begin
    u := rnd (s); i := 1;
    while table[i,1]<u do i := i+1;
    sample := table[i,2];
  end { sample };

{ histograms }
procedure reset_histogram (var h :histogram);
  var
    i :cell_num;
  begin
    with h do
      begin
        total := 0; sosq := 0; count := 0;
        min := maxint; max := 0; expended := tim;
        for i := 0 to max_cell_num do cell[i] := 0;
      end;
  end { reset histogram };

procedure make_histogram (var h :histogram;
                          cell_base,cell_width :real; s :boolean);
  begin
    with h do
      begin
        width := cell_width; base := cell_base; state := s;
      end;
    reset_histogram (h);
  end { make histogram };
```

```
procedure print_histogram (var pr :text; h :histogram;
                                    len :string_length);
  const
    rw      = 7;
    rd      = 2;
    asterix = '*';
    space   = ' ';
  var
    max_value,m,v :real;
    i             :cardinal;

  procedure lines;
    var
      line :a_string;
      i,k  :cardinal;
    begin
      with h do
        for i :=  0 to max_cell_num do
          begin
            line := '';
            for k := 1 to trunc (len*cell[i]/max_value)
              do line := concat (line,asterix);
            while length (line)<len do
              line := concat (line,space);
            if i = 0 then write (pr,'under  ')
              else
                if i = max_cell_num then write (pr,'over   ')
                  else write (pr,base+((i-1)*width) :rw:rd);
            writeln (pr,'  ',line,cell[i] :rw:rd);
          end;
    end { lines };
begin
  with h do
    if count = 0 then sim_error ('histogram is empty')
    else
      begin
        max_value := 0;
        for i := 0 to max_cell_num do
          if cell[i]>max_value then max_value := cell[i];
        m := total/count;
        if count>1 then v := (sosq-total*m)/(count-1)
          else v := 0;
        if state then
          write (pr,'processed =  ',trunc(count) :5);
        writeln (pr);
        writeln (pr,'mean =  ',m :rw:rd,'  variance =  ',
                 v :rw:rd,'  sd =  ',sqrt(v) :rw:rd);
        writeln (pr,'min =  ',min :rw:rd,'  max =  ',
                 max :rw:rd);
        writeln (pr);
        lines;
      end;
end { print histogram };
```



```
procedure log_histogram (var h :histogram; x,y :real);
  var
    i :cell_num;
  begin
    with h do
      begin
        i := trunc ((x-base)/width+1);
        if i<0 then i := 0;
        if i>max_cell_num then i := max_cell_num;
        if x>max then max := x;
        if x<min then min := x;
        if not state then
          begin
            y := tim-expended; expended := tim;
          end;
        cell[i] := cell[i]+y; total := total+y*x;
        sosq := sosq+x*x*y; count := count+y;
      end;
  end { log histogram };

{ screen control }
procedure make_screen;
  const
    ESC = 27;
  begin
    write (chr(ESC),'[=3h');
  end { make screen };

procedure gotoxy (x,y :cardinal);
  const
    ESC = 27;
  begin
    write (chr(ESC),'[',y-1 :1,';',x-1 :1,'H');
  end { gotoxy };

procedure clear_screen;
  const
    ESC = 27;
  begin
    gotoxy (0,0);
    write (chr(ESC),'[2J');
  end { clear screen };
```

```
procedure set_foreground (c :color);
  { set for extended ANSI }
  const
    ESC = 27;
  begin
    write (chr(ESC));
    case c of
      black :write ('[30m');
        red :write ('[31m');
      green :write ('[32m');
     yellow :write ('[33m');
       blue :write ('[34m');
    magenta :write ('[35m');
       cyan :write ('[36m');
      white :write ('[37m');
    end;
  end { set foreground };

procedure set_background (c :color);
  { set for extended ANSI }
  const
    ESC = 27;
  begin
    write (chr(ESC));
    case c of
      black :write ('[40m');
        red :write ('[41m');
      green :write ('[42m');
     yellow :write ('[43m');
       blue :write ('[44m');
    magenta :write ('[45m');
       cyan :write ('[46m');
      white :write ('[47m');
    end;
  end { set background };

procedure set_foreground (c :color);{ Turbo Pascal version }
  begin
    case c of
      black :textcolor (0);
        red :textcolor (4);
      green :textcolor (2);
     yellow :textcolor (14);
       blue :textcolor (1);
    magenta :textcolor (5);
       cyan :textcolor (3);
      white :textcolor (15);
    end;
  end { set foreground };
```



```
procedure set_background (c :color);{ Turbo Pascal version }
  begin
    case c of
      black :textbackground (0);
        red :textbackground (4);
      green :textbackground (2);
     yellow :textbackground (14);
       blue :textbackground (1);
    magenta :textbackground (5);
       cyan :textbackground (3);
      white :textbackground (15);
    end;
  end { set background };

procedure reset_colors;
  begin
    set_background (black);
    set_foreground (white);
  end { reset colors };

{ visual display }
procedure delay;
  var
    i :cardinal;
  begin
    for i := 1 to delay_num do write ('');
  end { delay };

procedure make_class_table;
  var
    i :class_num;
  begin
    for i := 1 to max_class_num do
      with class_table[i] do
        begin
          let := 'a'; col := white;
        end;
  end { make class table };

procedure enter_class (n :class_num; l :char; c :color);
  begin
    with class_table[n] do
      begin
        let := l; col := c;
      end;
  end { enter class };
```

```
procedure write_entity (x,y :cardinal; e :entity);
  begin
    gotoxy (x,y);
    with e^ do
      begin
        if col<>nul then set_foreground (col)
          else set_foreground (class_table[class].col);
        write (class_table[class].let);
      end;
  end { write entity };

procedure write_queue (x,y :cardinal; b :color;
                       q :queue; max_length :cardinal);
  const
    space = ' ';
  var
    xx :integer;
    p  :link;
  begin
    set_background (b);
    xx := x; p := q^.next;
    while (p<>q) and (xx>0) and (x-xx<max_length) do
      begin
        write_entity (xx,y,p^.item);
        xx := xx-1; p := p^.next;
      end;
    while (xx>0) and (x-xx<max_length) do
      begin
        gotoxy (xx,y); write (space);
        xx := xx-1;
      end;
    reset_colors;
    delay;
  end { write queue };

procedure write_block (x1,y1,x2,y2 :cardinal; b :color);
  const
    space = ' ';
  var
    i,j :cardinal;
  begin
    set_background (b);
    for i := x1 to x2 do
      for j := y1 to y2 do
        begin
          gotoxy (i,j); write (space);
        end;
    reset_colors;
  end { write block };
```



```
procedure move_v (x,y1,y2 :cardinal; e :entity; b :color);
  const
    space = ' ';
  var
    i :integer;
  begin
    if y1 >= y2 then sim_error ('moving vertical entity')
      else
        begin
          set_background (b); write_entity (x,y1,e);
          for i := y1+1 to y2 do
            begin
              delay;
              write_entity (x,i,e);
              gotoxy (x,i-1); write (space);
            end;
          gotoxy (x,y2); write (space);
          reset_colors;
        end;
  end { move vertical entity };

procedure move_h (y,x1,x2 :cardinal; e :entity; b :color);
  const
    space = ' ';
  var
    i :integer;
  begin
    if x1 >= x2 then sim_error ('moving horizontal entity')
      else
        begin
          set_background (b);
          write_entity (x1,y,e);
          for i := x1+1 to x2 do
            begin
              delay;
              write_entity (i,y,e);
              gotoxy (i-1,y); write (space);
            end;
          gotoxy (x2,y); write (space);
          reset_colors;
        end;
  end { move horizontal entity };

procedure write_time;
  var
    h :cardinal;
  begin
    h := trunc (tim) mod (24*60);
    write (trunc (tim) div (24*60) :3,' :',h div 60 :2,' :',
           (h mod 60)+ (tim-trunc (tim)) :5:2);
  end { write time };
```

```
{ user written procedures }
procedure display;
procedure initialize;
procedure picture;
procedure report;

{ simulation executive - three phase and event }
procedure run (duration :real; max_C :cardinal);
  var
    c :cardinal;
  begin
    running := true;
    repeat
      if calendar=calendar^.next then running := false
        else
          begin
            display;
            tim := calendar^.next^.item^.time;
            if duration<tim then running := false
              else
                begin
                  while (calendar<>calendar^.next) and
                      (tim=calendar^.next^.item^.time)  do
                    begin
                      calendar_top;
                      case current^.next_B of
                        0 : ;
                      end;
                    end;
                  for c := 1 to max_C do
                    case c of
                      1 : ;
                    end;
                end;
          end
    until not running;
  end { run - three-phase and event };

procedure run (duration :real);  { process view }
  var
    c       :link;
    changed :boolean;
    e       :entity;
    present :cardinal;
  procedure reactivate;
    begin
      on_calendar := false;
      present := current^.next_B;
      case current^.class of
        0 : ;
      end;
    end (reactivate);
```



```
    begin
      running := true;
      repeat
        if calendar = calendar^.next then running := false
          else
            begin
              display;
              tim := calendar^.next^.item^.time;
              if duration<tim then running := false
                else
                  begin
                    while (calendar<>calendar^.next) and
                          (tim=calendar^.next^.item^.time)  do
                      begin
                        calendar_top;
                        reactivate;
                        if current^.next_B = 0 then dis_entity
                        (current)
                          else
                            begin
                              if not on_calendar then
                                give_tail
                                (suspended_chain,current);
                            end;
                      end;
                    repeat
                      changed := false;
                      c := suspended_chain^.next;
                      while c<>suspended_chain do
                        begin
                          current := c^.item;
                          reactivate;
                          c := c^.next;
                          if on_calendar or
                          (present<>current^.next_B) then
                            begin
                              changed := true;
                              e := take (suspended_chain,c^.pre);
                            end;
                          if current^.next_B = 0 then dis_entity
                          (current)
                            else
                              if (not on_calendar) and
                                (present<>current^.next_B) then
                                  give_tail
                                  (suspended_chain,current);
                        end
                    until not changed;
                  end;
            end
      until not running;
    end { run };
```

Appendix C

Implementing Pascal_SIM

The Pascal_SIM facilities listed in Appendix B need to be implemented by a particular system (i.e., computer, operating system, Pascal compiler and run-time system). This generally involves two steps: slight alteration to the routines, particularly to the screen control codes, followed by the storage of the routines in a library or linkable module so that they do not have to be recompiled each time with the simulation program.

General implementation points are, followed by detailed implementation instructions for five different Pascal systems:

(*a*) Borland's Turbo Pascal 3 running under PC-DOS,
(*b*) Prospero's Pro Pascal 2 running under MS-DOS,
(*c*) DEC VAX/VMS Pascal running under VMS,
(*d*) UCSD Pascal,
(*e*) Sorrento Valley Associates' (SVS) Pascal running under UNIX.

These were chosen simply because the authors have experience of implementing Pascal_SIM with these systems. However, similar principles will apply to other Pascal systems.

If you are implementing Appendix B from the text, it may be useful to implement Pascal_SIM without the visual facilities, at least initially (See Appendix D). The procedures and functions under the headings screen control and visual displays are left out which saves typing, since they account for about 50 % of the total amount of code.

C.1 POSSIBLE ALTERATIONS

C.1.1 Random number generator

As the pseudo-random number generator is at the heart of any simulation program, the implementation of a sound generator is fundamentally important. While many Pascal implementations will provide a built-in generator, frequently these cannot be seeded, and thus any sequence of random numbers generated cannot be repeated, or they are of poor quality.

Two generators are provided in Appendix B, one for 16-bit machines, and

the other for 32-bit machines. The 16-bit generator uses the overflow occurring with the modulo operation. It works quite well with Turbo Pascal (version 2·0 or 3·0), and can be used with UCSD Pascal by disabling overflow detection with a compiler directive.

The 32-bit generator works well on any 32-bit machine, provided that the constants *mult* and *divid* can be stored with complete accuracy. Frequently, this will mean altering *mult* and *divid* to be double-precision constants, and using double-precision arithmetic for the statement:

quotient := *r*/*divid*;

i.e., *quotient* and *r* must be declared as double-precision.

C.1.2 Screen control

The procedures:

make_screen
gotoxy
clear_screen
set_foreground (two versions)
set_background (two versions)

contain all the necessary screen control codes. The codes in the first three procedures and in the first versions of the other two are set to the extended ANSI color codes used by the IBM-PC. They should also work with any black and white ANSI terminal, for example the VT100, but of course no color will be obtained. The second versions of **set_foreground** and **set_background** are for use with Turbo. If you are using a Pascal with existing **gotoxy**, **make_screen** and **clear_screen** functions or their equivalents (for instance Turbo or UCSD Pascal), then use those instead.

C.1.3 New_entity

The procedures and functions are grouped in a logical order. However, this means that the declaration of **new_entity** follows its use in **make_queue**. Thus depending on the method of implementation, **new_entity** might have to be forward declared or moved to a position preceeding **make_queue**. This is certainly necessary for Turbo, Pro, and VAX/VMS Pascal, but unnecessary for UCSD or SVS Pascal, where all procedure and function headings have to be declared separately in an interface area.

If the entity type is changed to provide additional fields or variant records, **new_entity** also needs to be altered. The latter is explained in Appendix 7.A. A change to the entity type means that Pascal_SIM has to be altered, and also recompiled if it is in a separated pre-compiled module. It is thus tailored to the individual simulation program.

C.1.4 Other alterations

Other alterations may be necessary. Some Pascal implementations do not allow the under-score characters, so they will have to be edited out. Strings are implemented differently in different Pascal implementations (although there is a *de facto* standard which has descended from UCSD Pascal), and so some alterations may be necessary (this is true for VAX/VMS Pascal).

The various constants at the start of Pascal_SIM, may be altered as necessary. These are:

max_cell_num	number of cells in a histogram
max_stream_num	number of available random number streams
max_class_num	number of different entity classes allowed
max_sample_num	number of empirical distributions allowed
max_string_length	size of strings in error messages
delay_num	the dummy delay used by visual routines

The constant **delay_num**, in particular, will have to be altered so that visual displays on a specific system advance at the desired rate.

C.2 SPECIFIC PASCAL SYSTEMS

C.2.1 Turbo Pascal 3

Turbo Pascal 3 is an excellent medium for developing visual simulations quickly and easily. However, the poor precision of the real numbers and the lack of accuracy in the trigonometric functions, may lead to poor quality variate generation.

The IBM-PC version of Turbo Pascal has some built in visual display routines which can be used in addition to those in Pascal_SIM. However, Turbo Pascal 3 provides no facilities for establishing libraries or separately compiling modules, thus Pascal_SIM must be included and compiled directly with the simulation program. To prepare for use with Turbo Pascal do the following.

(*a*) Use the 16-bit random number generator (delete the 32-bit generator).
(*b*) Delete the routines **make_screen**, **gotoxy** and **clear_screen**. In their place, use the built in Turbo Pascal functions *crtinit*, *gotoxy* and *clrscr*.
(*c*) Use the special Turbo Pascal versions of **set_foreground** and **set_background** (delete the others), which translate calls into calls to the built in Turbo routines *textcolor* and *textbackground*.
(*d*) Turbo Pascal provides a built in delay procedure *delay* (*time :integer*), where *time* is in milliseconds. If desired, this can be used in place of the Pascal_SIM **delay** procedure used by the visual routines.

To implement with Turbo Pascal either use the Pascal_SIM constant, type and variable declarations, and routines directly in the text of the simulation program or collect them into a separate file and access the file as an *include* file.

C.2.2 Pro Pascal 2

Prospero's Pro Pascal is a high-quality optimizing compiler for MS-DOS and CP/M machines. While the three-pass compiler is comparatively slow, and link times can also be large, the resulting code is fast and real number precision is good. Thus it is a good medium for running simulation experiments.

Few alterations are necessary and the Pascal_SIM routines can be compiled as a separate module. To prepare for use with Pro Pascal do the following.

(*a*) Delete the 16-bit machine generator. The 32-bit generator will work with Pro Pascal if the constants *divid* and *mult* are declared as integer (since *maxint* in Pro Pascal is 2147483647), and *r* and *quotient* are declared as *longreal* types (*longreal* is the Pro Pascal double precision real type). Thus use as a generator:

```
function rnd (s :stream_num) :real;
{ generator for 32-bit integers }
  const
    mult = 16807;
    divid = 2147483647;
  var
    r,quotient :longreal;
  begin
    r := mult*seeds[s];
    quotient := r/divid;
    seeds[s] := trunc (r-(trunc(quotient)*divid));
    rnd := seeds[s]/divid;
  end { random number generator };
```

(*b*) Delete the versions of **set_foreground** and **set_background** for Turbo Pascal.
(*c*) Pro Pascal provides a built-in keyboard polling function called *cstat*, which returns true when a key is pressed. (This is useful for programming user-determined interaction.)

To implement with Pro Pascal:

(*a*) Prepare a file containing all of Pascal_SIM called *PSIM.PAS*. Declare all Pascal_SIM variables to be *common*. (This is a Pro Pascal extension that allows a program to share variables with a module). This is done by using *common* in place of *var*.
(*b*) Compile Pascal_SIM as a Pro Pascal segment. This is done by starting *PSIM.PAS* with

```
segment Pascal_SIM;
```

(where Pascal_SIM is the name given to the segment)

as the header line, and a dummy main program body thus:

```
begin
end.
```

The result of the compilation is an object file called *PSIM.OBJ*.

(*c*) Now within any simulation program:

(i) the Pascal_SIM constants and types will have to be newly declared alongside the program constants and types,
(ii) the Pascal_SIM variables must be declared as *common*; any program variables can be declared as *var*,
(iii) all Pascal_SIM functions and procedures must be declared as *external*.

(*d*) The compiled program must be linked with *PSIM.OBJ*, and the Pro Pascal standard library. This must be done with the Prospero linker, e.g.,

```
prolink sim,psim,paslibs/s
```

where *sim* is a previously compiled simulated program, and */s* means that the standard library is selectively scanned. An executable file *sim.exe* is produced.

(*e*) Note that on an IBM-PC it is necessary to prepare PC-DOS for extended ANSI screen control by installing the appropriate screen driver. This is done by adding the line

```
device = ansi.sys
```

to the *config.sys* file, and putting the provided driver *ansi.sys* on the boot disk.

C.2.3 VAX/VMS Pascal

Those with access to a DEC VAX computer will probably have access to VAX/VMS Pascal. Implementation in VAX/VMS Pascal is fairly painless, although VAX/VMS Pascal handles strings rather differently from other Pascal implementations. A good module facility is provided; Pascal_SIM can be compiled and established so that it does not have to be compiled with the simulation program. Pascal_SIM can be implemented easily on some black and white terminals, such as a VT100, although you may wish to remove all references to color, other than black and white before you start. Other types of terminals need different strings of characters for screen control. However, there is no problem in implementing the simple version of Pascal_SIM shown in Appendix D.

The rest of this section will assume you have a VT100, or similar terminal, available for use. To prepare for use with VAX/VMS Pascal note the following.

(*a*) The string type is implemented in VAX/VMS Pascal as *varying [...] of char*. Thus the type declaration

```
a_string = string [max_string_length];
```

is replaced with

```
a_string = varying [max_string_length] of char;
```

(*b*) The concatenation of strings is achieved with the plus operator, rather than the *concat* function. Thus where


```
line := concat (line,asterix);
```

appears in **print_histogram**, it must be replaced with

```
line := line+asterix;
```

(*c*) The **make_screen** procedure should be removed (it is unnecessary when using DEC terminals).

(*d*) The constant **delay_num** will almost certainly have to be reduced in size if Pascal_SIM is being implemented for a multi-user VAX. A good guess at an appropriate value is 100. (However, if the VAX is heavily loaded, you might as well set **delay_num** to be zero, since the terminal polling will negate the effect of trying to neatly time slice the display.)

(*e*) The 32-bit generator should be used (delete the 16-bit generator). However, single precision arithmetic in VAX/VMS Pascal is unsatisfactory. Alter the generator to:

```
function rnd (s :stream_num) :real;
{ generator for 32-bit integers }
  const
    mult = 16807.0d0;
    divid = 2147483647.0d0;
  var
    r,quotient :double;
begin
  r :=mult*seeds[s];
  quotient := r/divid;
  seeds[s] := trunc(r-(trunc(quotient)*divid));
  rnd := sngl (seeds[s]/divid);
end { random number generator };
```

where *double* is the VAX/VMS double-precision type, and *sngl* is a built-in function that converts double-precision back to single-precision.

(*f*) Delete the versions of **set_foreground** and **set_background** for Turbo Pascal.

(*g*) VMS stores output in a buffer, which is emptied when a *writeln* is issued. Thus a visual simulation that only uses **gotoxy** and *write* will simply fill up the buffer. Eventually, a 'buffer full' error message will occur. This can be overcome by making **gotoxy** do a dummy *writeln* thus:

```
procedure gotoxy (x,y :cardinal);
  const
    esc = 27;
  begin
    writeln;
    write (chr(esc),'[',y-1:1, ';', x-1:1, 'H');
  end { gotoxy };
```

To implement with VAX/VMS Pascal do the following.

(*a*) Prepare a module file called PSIM.PAS containing all of Pascal_SIM

(the constant, type, and variable declarations, plus all procedures and functions). The file should have

[environment ('psim.pen')] module Pascal_SIM (output);

as a header line (where Pascal_SIM is the name given to the module), and finish with a single

end.

which terminates the module. Each outer-level function and procedure should be prefaced by the qualifier *[global]*, e.g.,

[global] procedure **make_sim;**

This allows Pascal_SIM to be used by a simulation program simply by inheriting an environment file, and linking with it any following compilation.

(*b*) Compile *psim.pas*. This produces an environment file *psim.pen*, and an object file containing the routines called *psim.obj*.

(*c*) Any simulation program can now inherit the environment, e.g.,

[inherit ('psim.pen')] program simulation;

(*d*) The environment file must be linked with the compiled simulation program, e.g.,

link sim,psim

where *sim* is a previously compiled simulation program.

C.2.4 UCSD Pascal

While perhaps not as popular as it was, UCSD Pascal is one of the best implementations of Pascal available. (It appears on the Apple II as Apple Pascal.)

UCSD Pascal provides an excellent means of implementing collections of constants, types, variables, and routines in a library from where they can be used by any program, called the unit. To prepare for use with UCSD Pascal note the following.

(*a*) On a 16-bit machine, the 16-bit generator will work if detection of integer overflow can be disabled. This can be done by disabling range checking with a

{$R-}

compiler flag placed at the start of the **rnd** procedure. On 32-bit machines, check the precision of the particular UCSD implementation before implementing the 32-bit generator.

(*b*) Delete the versions of **set_foreground** and **set_background** for Turbo Pascal.

(*c*) Delete the Pascal_SIM version of **gotoxy**, because UCSD Pascal provides a built in *gotoxy* procedure.

(*d*) UCSD Pascal provides a built in keyboard polling function called *keypress*, which returns *true* when a key is pressed. (This is useful for programming user-determined interaction.)

To implement with UCSD Pascal note the following.

(*a*) Pascal_SIM must be prepared as an intrinsic unit for insertion in the library. Prepare a text file of the form:

```
unit Pascal_SIM;
intrinsic code 23 data 24;
interface
  <A>
implementation
  <B>
  begin
  end.
```

where A contains the Pascal_SIM constant, type, and variable declarations, and all function and procedure heads; B contains all function and procedure bodies but no parameter declarations as these have already appeared in the unit interface. Slots in the library must be specified by number – use any two slots that are not in use (23 and 24 are usually available).

(*b*) The unit can be compiled, and put into the UCSD system library using the library utility.

(*c*) Any program can now use Pascal_SIM by using the UCSD specification:

uses Pascal_SIM;

immediately after the program heading.

(*d*) Later versions of UCSD Pascal (version 4 onwards) allow for function and procedure calls to be placed in the dummy main program body. These are then invoked when a program that uses the unit is executed. Thus calls to **make_screen**, **make_ streams** and **make_sim** can be put here, rather than in the simulation program.

C.2.5 SVS Pascal

SVS Pascal runs under UNIX on a number of UNIX workstations and minicomputers. It provides a unit facility similar to UCSD Pascal, and close adherence to ISO Pascal. To prepare for use with SVS Pascal note the following.

(*a*) The 32-bit generator should be used (delete the 16-bit generator). Double-precision arithmetic should be used thus:

```
function rnd (s :stream_num) :real;
{ generator for 32-bit integers }
  const
    mult = 16807.0d0;
    divid = 2147483647.0d0;
  var
    r,quotient :long;
begin
  r := mult*seeds[s];
  quotient := r/divid;
  seeds[s] := trunc (r-(trunc(quotient)*divid));
  rnd := seeds[s]/divid;
end { random number generator };
```

where *long* is the SVS double-precision type.

(*b*) As *maxint* is 32768 in SVS Pascal, and seeds need to cover the entire range of the divisor *divid*, the SVS type *longint* must be used. Both the seeds and the original seeds should be declared as an array of *longint*. Alternatively, **cardinal** can be declared as 0..2147483647.

(*c*) Delete the versions of **set_foreground** and **set_background** for Turbo Pascal.

Appendix D

Pascal_SIM without Visual Facilities

This appendix shows the declarations, procedures, and functions needed for Pascal_SIM non-visual simulations. The **new_entity** procedure, which is different from the one in Appendix B, is shown in full.

```
const
  max_cell_num      = 16;
  max_stream_num    = 32;
  max_class_num     = 256;
  max_sample_num    = 20;
  max_string_length = 80;

type
  a_string = string [max_string_length];
  cardinal = 0..maxint;
  stream_num    = 1..max_stream_num;
  cell_num      = 0..max_cell_num;
  class_num     = 1..max_class_num;
  sample_num    = 1..max_sample_num;
  string_length = 1..max_string_length;
  entity = ^an_entity;
  link   = ^a_link;
  a_link = record
             next,pre :link;
             item     :entity;
           end;
  queue  = link;
  an_entity = packed record
                avail       :boolean;
                class       :class_num;
                attr,next_B :cardinal;
                time        :real;
              end;
  bin = record
          number,num_avail :cardinal;
        end;
  histogram = record
                cell                    :array [cell_num] of real;
                count,width,base,
                  total,sosq,
                  min,max,expended :real;
                state                   :boolean;
              end;

  lookup_table = array [sample_num,1..2] of real;
```

```
var
  tim                  :real;
  current              :entity;
  calendar             :queue;
  on_calendar          :boolean;
  suspended_chain      :queue;
  running              :boolean;
  original_seeds,seeds :array [stream_num] of cardinal;

{ available routines are

- error messages -
procedure sim_error (s :a_string);

- queue processing -
procedure make_queue (var q :queue);
procedure give (q :queue; t :link; i :entity);
function  take (q :queue; t :link) :entity;
procedure give_top (q :queue; i :entity);
procedure give_tail (q :queue; i :entity);
function  take_top (q :queue) :entity;
function  take_tail (q :queue) :entity;
function  empty (q :queue) :boolean;

- entities and classes -
function  new_entity (c :class_num; a :cardinal) :entity;
procedure dis_entity (e :entity);
function  count (q :queue) :cardinal;
procedure make_class (var c :queue; n,size :cardinal);

- timing and the executive -
procedure calendar_top;
procedure make_sim;
procedure cause (nb :cardinal; e :entity; t :real);

- facilities for process executive -
procedure branch (next :cardinal);
procedure remove_entity;

- resources -
procedure make_bin (var from :bin; n :cardinal);
procedure acquire (var from :bin; n :cardinal);
procedure return (var from :bin; n :cardinal);

- random number generation and streams -
procedure make_streams;
function rnd (s :stream_num) :real;
```



```
 - distribution sampling -
function normal (m,sd :real; s :stream_num) :real;
function log_normal (m,sd :real; s :stream_num) :real;
function poisson (m :real; s :stream_num) :cardinal;
function negexp (m :real; s :stream_num) :real;
function uniform (l,h :real; s :stream_num) :real;
procedure make_sample (var sample_file :text;
                       var table :lookup_table);
function sample (table :lookup_table; s :stream_num) :real;

- histograms -
procedure reset_histogram (var h :histogram);
procedure make_histogram (var h :histogram;
                          cell_base,cell_width :real; s :boolean);
procedure print_histogram (var pr:text;h:histogram;
                           len :string_length);
procedure log_histogram (var h :histogram; x,y :real);
procedure write_time;

- dummy procedures -
procedure display;
procedure initialize;
procedure picture;
procedure report;

- simulation executive -
procedure run (duration :real; max_C :cardinal); }

{ changed new_entity procedure }

function new_entity (c :class_num; a :cardinal) :entity;
  var
    e :entity;
  begin
    new (e);
    with e^ do
      begin
        time := 0.0; avail := true; class := c;
        col := nul; attr := a; next_B := 0;
      end;
    new_entity := e;
  end { new entity };
```

References

METHODOLOGY

Crookes, J.G., 1982, Simulation in 1981. *Eur. J. Op. Res.*, **9**, 1.
State of the art review, most of which is still relevant. Includes an argument for use of the three-phase approach, and a discussion of the value of visual output.

Davies, R.M., 1985a, An assessment of models of a health care system. *J. Ops. Res. Soc.*, **35**, 679.
A paper which assesses the benefits of simulation compared to other modelling techniques.

DeMarco, T., 1978, *Structured Analysis and System Specification*, Yourdon Press Computing Series (Englewood Cliffs, NJ: Prentice Hall).
A very readable book on structured systems analysis, with a chapter about using structured English to describe systems.

Keen, P.G.W., and Scott-Morton, M.S., 1978, *Decision Support Systems: An Organisational Perspective* (Reading, MA: Addison-Wesley).
A book about decision support systems.

Oren, T.I., and Zeigler, B.P., 1979, Concepts for advanced simulation methodologies. *Simulation*, **32**, 69.
A paper which describes a structured method, called an experimental frame, for separating the specification of experiments from the model.

Rivett, P., 1980, *Model Building for Decision Analysis* (New York: Wiley).
A theoretical book with a good first chapter on modelling methodology.

Roberts, N., Anderson, D., Deal, R., Garet, M., and Shaffer, W., 1983, *Introduction to Computer Simulation: A System Dynamics Modeling Approach* (Reading, MA: Addison-Wesley).
A comprehensive introductory text on systems dynamics.

Spinelli de Carvalho, R., 1976, Cellular simulation. Ph.D. thesis, University of Lancaster, England.
The Ph.D. thesis that generated the idea of cellular simulation.

Tocher, K.D., 1962, *The Art of Simulation* (London: English Universities Press).
The first text on discrete-event simulation. Discussion of methodology and modelling still relevant.

Zeigler, B.P., 1984, System-theoretic representation of simulation models. *IIE Trans.*, **16**, 19.
A paper describing the use of systems theory in simulation.

STATISTICS

Box, G.E.P., and Muller, M.E., 1958, A note on the generation of Normal deviates. *Ann. Math. Statist.*, **28**, 610.
A description of the Box–Muller method of sampling from a Normal distribution.

Cochran, W.G., and Cox, G.M., 1957, *Experimental Design*, 2nd edition, (New York: Wiley).
A standard text on the techniques of experimental design.

Fishman, G.S., 1978, *Concepts and Methods in Discrete Event Digital Simulation* (New York: Wiley).
A book with good coverage of statistical aspects of simulation, particularly random number generation.

Fishman, G.S., and Huang, B.D., 1983, Antithetic variates revisited. *Commun. ACM*, **26**, 964.
A mathematical analysis of antithetic variates showing how to use them to greatest advantage.

Fishman, G.S., and More, L.R., 1982, A statistical evaluation of multiplicative congruential random number generators with modulus $2^{31}-1$. *J. Am. statist. Ass.*, **77** (377), 129.
A paper describing of the statistical testing of 32-bit random number generators.

Freud, J.E., and Walpole, R.E., 1980, *Mathematical Statistics*, 3rd edition (Englewood Cliffs, NJ: Prentice Hall).
A good and readily comprehensible general statistics text book.

Gafarian, A.V., Ancker, C.J., and Morisaku, T., 1978, Evaluation of commonly used rules for detecting steady-state in computer simulation. *Naval Res. Log. Quart.*, **25**, 511.
A comparison of various methods, including the cumulative moving average method discussed in Chapter 5.

Hoel, P.G., 1976, *Elementary Statistics*, 4th edition (New York: Wiley).
A good basic statistics text; a useful starting point for confidence intervals and hypothesis testing.

IBM, 1969, Random number generation and testing. IBM publication number GC20-8011-0.
A paper describing the statistical testing of random numbers.

Klein, R.W., and Roberts, S.D., 1984, A time-varying Poisson arrival generator. *Simulation*, **43**, 193.

An inversion method for dealing with time-varying Poisson arrivals.

Knuth, D.W., 1969, *The Art of Computer Programming: Vol. 2, Semi-numerical* Algorithms (Reading, MA: Addison-Wesley).
The bible on random number generation.

Law, A.M., and Kelton, D., 1982, *Simulation Modeling and Analysis* (New York: McGraw-Hill).
An extensive text on all statistical aspects of simulation modelling. The material on distributions, distribution fitting, and random variate generation is extensive.

Law, A.M., and Vincent, S.G., 1986, A tutorial on UNIFIT: an interactive computer package for fitting probability distributions to observed data. *Proc. 1986 Winter Simulation Conf.*, edited by Wilson, Henriksen, and Roberts, IEEE Computer Society, pp. 218–222.
A description of a useful package for fitting distribution data.

Lewis, P.A., Goodman, A.S., and Miller, J.M., 1969, A pseudo-random number generator for the IBM 360. *IBM Systems J.*, **8**, 136.
Describes tests used on a 32-bit random number generator and recommends its use. This generator is used in PascaLSIM.

Lewis, P.A.W., and Shedler, G.S., 1979, Simulation of nonhomogeneous processes by thinning. *Naval Res. Log. Quart.* **26**, 403.
A theoretical explanation of the use of thinning for time-varying Poisson arrivals.

Rubinstein, R.Y., and Scmorodnitsky, G., 1985, Variance reduction by the use of common and antithetic random variates. *J. Statist. Comput. Simul.*, **22**, 161.
A comparison of the two methods.

Thensen, A., Sun, Z., and Wang, T., 1984, Some efficient random number generators for micro-computers. *Proc. 1984 Winter Simulation Conf.*, edited by Sheppard, Pooch, and Pegden, IEEE Computer Society, pp. 187–196.
Tested implementations of random number generators for microcomputers. The 16-bit generator provided here in Pascal was taken from this paper.

PASCAL AND PROGRAMMING

Barron, D.W., and Bishop, J.M., 1984, *Advanced Programming: A Practical Course* (New York: Wiley).
A book which goes beyond the basics of Pascal. Covers sorting and searching, data structures, handling files, etc. Very readable.

Crookes, J.G., Balmer, D.W., Chew, S.T., and Paul, R.J., 1986, A three-phase simulation model written in Pascal. *J. Ops. Res. Soc.*, **37**, 603.
An alternative Pascal package for three-phase simulations.

Findlay, W., and Watt, D.A., 1985, *Pascal, An Introduction to Methodological Programming* (London: Pitman).

A first book on Pascal, with a good explanation of the way pointers work.

Grogono, P., 1978, *Programming in Pascal* (Reading, MA: Addison-Wesley).
A book about Pascal which uses discrete-event simulation to explain the use of queues and linked lists.

Horowitz, E., and Sahni, S., 1976, *Fundamentals of Data Structures* (London: Pitman).
A classic text on data structures and associated algorithms. Covers queues and linked lists.

Kaubish, W.H., Perrott, R.H., and Hoare, C.A.R., 1976, Quasiparallel programming. *Software – Prac. and Exp.* **6**, 341.
A paper which presents SIMONE, an extension of Pascal for writing simulations, using a process interaction facility copied from SIMULA.

Kernighan, B.W., and Plauger, P.J., 1981, *Software Tools in Pascal* (Reading, MA: Addison-Wesley).
A book about good programming practice using Pascal.

Jennergren, L.P., 1984, *Discrete-Event Simulation Models in Pascal/MT+* (Bromley, UK: Chartwell-Bratt).
A short book on writing simulations from scratch in Pascal. Pascal/MT+ is an implementation for CP/M machines from Digital Research.

Jensen, K., and Wirth, N., 1985, *Pascal User Manual and Report*, 3rd edition (Berlin: Springer-Verlag).
Wirth's original text on Pascal, revised to conform to the ISO standard.

Uyenso, D., and Vaessen, W., 1980, PASSIM: a discrete-event simulation package for Pascal, *Simulation*, **35**, 183.
A Pascal-based package which implements the facilities of GPSS.

Walsh, J., and Elder, J., 1988, *Introduction to Pascal*, 3rd edition (London: Prentice Hall).
A good Pascal text including a chapter explaining the use of pointers.

Wilson, I.R., and Addyman, A.M., 1982, *A Practical Introduction to Pascal – with BS 6192* (London: Macmillan).
A book which includes the complete specification of the BS 6192 (ISO 7185) standard for Pascal. A useful reference for 'ISO standard' Pascal.

Wirth N., 1985, *Programming in Modula-2*, 2nd edition (Berlin: Springer-Verlag).
A good text on Modula-2, written by its creator.

SIMULATION PROGRAMMING LANGUAGES AND PACKAGES

Balci, O., 1986, Requirements for model development environments. *Comput. Ops. Res.*, **13**, 53.
A paper which discusses the need for, and a design for, a model development environment.

Birtwistle, G.M., 1979, *Discrete-Event Modelling in SIMULA* (London: Macmillan).
A highly readable text on DEMOS, a discrete-event package written in SIMULA. Some good examples.

Birtwistle, G.M., Dahl, O.J., Myhrhaug, B., and Nygaard, K., 1979, *SIMULA Begin* (Bromley, UK: Chartwell-Bratt).
The original text on SIMULA. Rather difficult to follow, but comprehensive.

Bryant, R.M., 1980, SIMPAS: a simulation language based on Pascal. *Proc. 1980 Winter Simulation Conf.*, edited by Oren, Shub, and Roth, IEEE Computer Society, pp. 25–40.
A Pascal-based package that implements the facilities of SIMSCRIPT.

Bryant, R.M., 1982, Discrete system simulation in Ada. *Simulation*, **39**, 111.
A paper which presents a process interaction package in Ada.

Clementson, A.T., 1978, ECSL. *Proc. 1978 UKSC Conf. on Computer Simulation*, IPC Press, Guildford, England.
A discussion of CAPS/ECSL, a program generator–language combination. ECSL employs activity scanning.

Goldberg, A., and Robson, D., 1983, *SMALLTALK-80: The Language and its Implementation* (Reading, MA: Addison-Wesley).
The text on SMALLTALK. Uses the development of a simulation package as a case study.

Hills, P.R., 1965, SIMON – a computer simulation language in Algol 60. *Digital Simulation in Operational Research*, edited by Hollingdale (London: English Universities Press).
Despite its age, still an interesting article. Clearly explains how to develop three-phase simulations in Algol 60.

Kreutzer, W., 1986, *System Simulation: Programming Styles and Languages* (Reading, MA: Addison-Wesley).
An extensive and delightful review of programming methods and available languages for discrete and continuous simulation. Some emphasis on SIMULA, plus a presentation of some utilities for programming simulations in Pascal.

Mathewson, S.C., 1984, the application of program generator software and its extensions to discrete-event simulation modeling. *IIE Trans.*, **16**, 3.
A good discussion of program generators, including the author's DRAFT system, which can generate programs in a number of different languages.

O'Keefe, R.M., 1987, An interactive simulation description interpreter. *Comput. Ops. Res.*, **14**, 273.
A presentation of the Inter_SIM package.

Poole, E., and Szymankiewicz, J.Z., 1977, *Using Simulation to Solve Problems* (London: McGraw-Hill).
A comprehensive, but outdated, text on the HOCUS package.

Pritsker, A.A.B., 1984, *Introduction to Simulation and SLAM II* (New York: Halsted Press).
A comprehensive text on SLAM II.

Roberts, S.D., 1983, *Simulation Modeling and Analysis with INSIGHT* (Indianapolis, IN: Regenstrief Institute).
The text on INSIGHT.

Russell, E.C., 1983, *Building Simulation Models with SIMSCRIPT II.5* (Los Angeles, CA: CACI).
A text which describes the use of SIMSCRIPT II.5.

Schriber, T.J., 1974, *Simulation Using GPSS* (New York: Wiley).
The classic text on GPSS.

Standridge, C.R., 1985, Performing simulation projects with the extended simulation system (TESS). *Simulation*, **45**, 283.
An introduction to TESS, a simulation development environment.

Tocher, K.D., 1966, Some techniques of model building. *Proc. IBM Scientific Comput. Symp. on Simulation Models and Gaming*, IBM, White Plains, NY, pp. 119–155.
A long presentation of GSP, which many consider to be the first ever simulation programming language.

VISUAL OUTPUT AND USER INTERACTION

Amiry, A.P., 1965, The simulation of information flow in a steelmaking plant. *Digital Simulation in Operational Research*, edited by Hollingdale (London: English Universities Press), pp. 347–356.
A description of a very early use of visual output.

Bell, P.C., 1985, Visual interactive modelling in operational research: successes and opportunities. *J. Ops. Res. Soc.*, **36**, 975.
A review of work in visual interactive modelling, including visual simulation.

Bell, P.C., and O'Keefe, R.M., 1987, Visual interactive simulation – history, recent developments and major issues. *Simulation*, **49**, 109.
A state of the art review, plus an application.

Ellison, D., Herschdorfer, I., and Tunnicliffe-Wilson, J., 1982, Interactive simulation on a microcomputer. *Simulation*, **38**, 161.
A discussion of the use of a microcomputer and the three-phase approach to produce visual output. Very readable.

Fiddy, E., Bright, J.G., and Hurrion, R.D., 1981, SEE-WHY: interactive simulation on the screen. *Proc. Inst. Mech. Engrs*, **c293–81**, 167.
A presentation of SEE-WHY, a seminal visual simulation package.

Hollocks, B., 1983, Simulation and the micro. *J. Ops. Res. Soc.*, **34**, 331.

A discussion of work in simulation at British Steel, and some presentation of FORSSIGHT, a visual simulation package.

Hurrion, R.D., 1980, An interactive visual simulation system for industrial management. *Eur. J. Op. Res.*, **5**, 86.
A presentation of some work in visual simulation and user interaction.

Pegden, L.A., Miles, T.I., and Diaz, G.A., 1985, Graphical interpretation of output illustrated by a SIMAN manufacturing simulation model. *Proc. 1985 Winter Simulation Conf.*, edited by Gantz, Blais, and Soloman, IEEE Computer Society, pp. 244–251.
A discussion by the developers of the SIMAN–CINEMA system of their ideas and the use of graphics and animation.

APPLICATIONS

Christy, D.P., and Watson, H.J., 1983, The application of simulation: a survey of industry practice. *Interfaces*, **13**, 47.
A review of the use of simulation in United States industry.

Davies, R.M., 1985b, An interactive simulation in the health service. *J. Ops. Res. Soc.*, **36**, 597.
A descripition of the modelling of the treatment of patients with end-stage renal failure.

Davies, H., and Davies, R.M., 1987, A simulation for planning renal services in Europe. *J. Ops. Res. Soc.*, **38**, 693.
A paper showing the problems of modelling a large health care system with a large database. Pascal pointers and entity records are used to provide entity characteristics and shadows, as explained in Chapter 13.

Macintosh, J.B., Hawkins, R.W., and Shepard, C.J., 1984, Simulation on microcomputers – the development of a visual interactive modelling philosophy. *Proc. 1984 Winter Simulation Conf.*, edited by Sheppard, Pooch, and Pegden, IEEE Computer Society.
A discussion of the use of simulation, and the success of visual simulation, at Ford of Europe.

O'Keefe, R.M., and Davies, R.M., 1986, A microcomputer system for simulation modelling. *Eur. J. Op. Res.*, **24**, 23.
A discussion on the use of Pascal, the three-phase approach, and a microcomputer.

ARTIFICIAL INTELLIGENCE AND SIMULATION

Baskaran, V., and Reddy, R., 1984, An introspective environment for knowledge based simulation. *Proc. 1984 Winter Simulation Conf.*, edited by Sheppard, Pooch, and Pegden, IEEE Computer Society, pp. 645–651.
A discussion of the automatic analysis of simulation in KBS (see Reddy *et al.* 1986, below).

Bruno, G., Elia, A., and Laface, P., 1986, A rule-based system to schedule production. *Computer*, **19**, 32.
A description of the use of activity scanning to produce a simulation in OPS5, a rule-based language.

Doukidis, G., and Paul, R., 1985, Research into expert systems to aid simulation model formulation. *J. Ops. Res. Soc.*, **36**, 319.
The use of expert systems and natural language to capture descriptions of simulations.

Evans, J.B., 1980, Simulation and intelligence. Technical report TR-A5-84, Centre of Computer Studies and Applications, University of Hong Kong.
A discussion of the need to model intelligent behavior in discrete simulation.

Fraught, W.S., 1986, Applications of AI in engineering. *Computer*, **19**, 17.
A discussion of SimKit, a simulation package developed in the knowledge engineering environment (KEE).

Futo, I., and Szeredi, J., 1982, A discrete simulation system based on artificial intelligence techniques. *Discrete Simulation and Related Fields*, edited by Javor (Amsterdam: North-Holland), pp. 135–150.
A readable paper on T-Prolog, a logic-based discrete simulation language developed in Hungary.

Haddock, J., 1987, An expert system framework based on a simulation generator. *Simulation*, **48**, 45.
An intelligent front end for simulations of manufacturing systems using the SIMAN language.

Klahr, P., Ellis, J.W., Giarla, W.D., Narain, S., Cesar, E.M., and Turner, S.R., 1986, TWIRL: tactical warfare in the ROSS language. *Expert Systems Techniques, Tools, and Applications*, edited by Klahr and Waterman (Reading, MA: Addison-Wesley), pp. 224–273.
An application of ROSS.

McArthur, D.J., Klahr, P., and Narain, S., 1986, ROSS: An object-oriented language for constructing simulations. *Expert Systems Techniques, Tools, and Applications*, edited by Klahr and Waterman (Reading, MA: Addison-Wesley), pp. 70–94.
Object and rule oriented simulation using LISP.

Minsky, M., 1977, Frame-system theory. *Thinking*, edited by Johnston-Laird and Wason (Cambridge, UK: Cambridge University Press), pp. 355–376.
Discovery of object-oriented representation by the artificial intelligence community.

O'Keefe, R.M., 1986, Simulation and expert systems: a taxonomy and some examples. *Simulation*, **46**, 10.
A discussion of the relationship between simulation and expert systems.

O'Keefe, R.M., Roach, J.W., 1987, Artificial intelligence approaches to simulation. *J. Ops. Res. Soc.*, **38**, 713.
An overview of the use of AI in building simulation tools, and a presentation of PROSS, a Prolog-based simulation language which combines the process description structure of GPSS with logic programming.

Reddy, Y.V.R., Fox, M.S., Husain, N., 1986, The knowledge-based simulation system. *IEEE Software*, **3**, 26.
The best of the papers that present the knowledge-based simulation system KBS.

Robertson, P., 1986, A rule-based expert simulation environment. *Intelligent Simulation Environments*, edited by Luker and Adelsberger (San Diego, CA: Society for Computer Simulation) pp. 9–15.
An interesting approach to building an advanced rule-based simulation tool.

Ruiz-Mier, S., and Talavage, J., 1987, A hybrid paradigm for modeling of complex systems. *Simulation*, **48**, 135.
An excellent presentation of SIMYON, a simulation language that combines an object-oriented approach with logic programming ideas.

Index