Software Engineering and Simulation

Kevin T. Ryan

Department of Computer Science
University of Kansas
Lawrence, Ks. 66045

Abstract

A tutorial survey is made of some techniques in Software Engineering and their relevance to simulation modelling. Various topics are discussed with examples.

In the area of Systems Design the topics are concurrency, deadlock and input verification; in the area of Program Design they are modular programming, stepwise refinement and the Jackson method; in the area of Program Verification, structured coding and program tracing and in the area of System Optimization, determining storage needs and monitoring CPU usage.

INTRODUCTION

For many people, the implementation of a simulation model as a computer program is their first large software project. It is often a painful introduction to the difference between small and large scale programming. As many who use computers in this way are not computer scientists, they may not be familiar with recent developments that would assist them in effectively designing and implementing simulation programs. This paper is a tutorial introduction to a range of methods and techniques that together form a large part of Software Engineering. In particular, it addresses the topics of System Design, Program Design, Program Verification and System Optimization. Methods and techniques are described which may be unknown to the novice, or experienced, simulation practitioner, and their relevance is demonstrated by examples from practical simulations.

SYSTEM DESIGN

The design of both applications and operating systems has been formalized to some extent. Techniques that were developed in these areas are useful to the designers of simulation models. The operating systems area has produced powerful notation for describing and controlling concurrent processes. Simulation designers can usefully study the "cobegin-coend" structure invented by Dijkstra [1] and further developed by Brinch-Hansen [2]. Hoare's monitor [3] allows us to describe controlled access to a critical resource, by a set of concurrent processes. Furthermore, as operating systems have become more concerned with resource sharing, techniques have been developed to describe and allocate resources. Shaw [4] is a good introduction to resource descriptors. Deadlock problems, where a system can no longer progress because of mutually irreconcilable needs, are a major concern particularly in multiprocessor systems. These problems can also arise in simulation models and an awareness of them will assist in model development. For example, a test for "dormant system" can be incorporated within the main event loop of an event-driven system. A useful survey of deadlock problems is Coffman, Elphick and Shoshani [5], while Holt [6] gives a more theoretical treatment.

Applications systems design is also relevant to simulation. Simulation models resemble applications systems in having somewhat arbitrary boundaries. Functional or corporate factors are usually decisive in applications areas whereas in simulation models the definition of the model boundary is usually a trade-off between accuracy and cost. The greater the degree of detail to be modelled the more precise and expensive will be the input data required. In fact the arbitrary nature of the boundaries of simulation models is recognized by our incorporation of stochasticity. Random inputs represent the uncertainty of our analysis of the real world. (Mihram [7] comments fully on this.) Deterministic or observed inputs represent the part of the system environment that we understand. At the systems design stage we can consider enclosing our model in what Blanning [8] calls a meta-model, and this meta-model may then provide most of the system inputs, whether random or not. An optimization algorithm could be incorporated at this level. For example, consider an urban bus route model where total cost of operating the bus system was stated as

CH1437-3/79/0477-0482$00.75 © 1979 IEEE
1979 Winter Simulation Conference

$$T_c = n_b \times B_c + n_p \times w_p \times w_c \qquad (1)$$

where T_c is the total cost of the system

n_b is the number of buses in operation

B_c is the cost of operating one bus

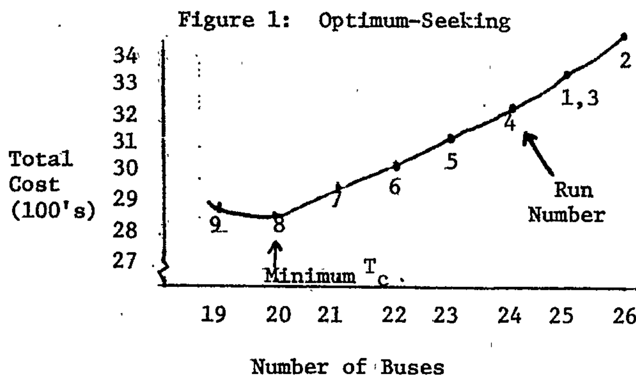
n_p is the number of passengers boarded

w_p is the average waiting time per passenger (in minutes)

w_c is the (attributed) cost of one minute's waiting

A model of this type was implemented and sought to minimise total cost (T_c) by varying only the number of buses in operation. Typical output is shown in Figure 1.

Figure 1: Optimum-Seeking

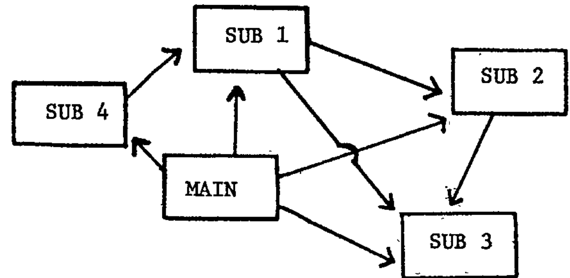


One final topic in systems design that is frequently omitted is the design of reliable input methods. If large quantities of computer resources are to be used in running a model, it is wise to perform the most thorough possible checks on the input data. After all, if the model is successful, it will be widely used. Unless a model builder has substantial experience of commercial data preparation, he or she is advised to read a good survey of methods and techniques for collecting clean and accurate data, for example Gilb [9] or Gilb and Weinberg [10]. Data collection forms should be designed to provide accurate data by considering the problem of data checking from the outset.

PROGRAM DESIGN

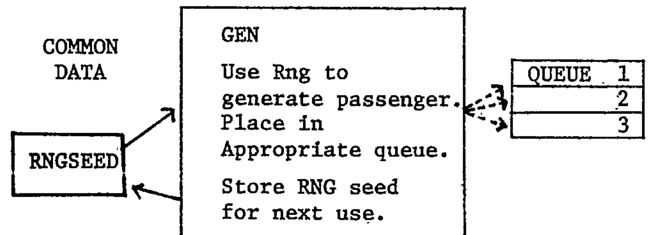
Most simulation programs have three top level functions, input validation, simulation process, and output generation. While the input and output modules may be complex, especially if extensive error checking and/or output selection are to be provided, most of the difficulty of program design lies in the simulation procedure. Designing large programs is qualitatively different from designing small programs. Realization of this in the 1960's led to the search for better methods. Modular programming was one of the first of these, but it could lead to overly complex programs. For example, as modular programming imposed no hierarchical structure on the overall program they could result in complex calling relationships as in Figure 2.

Figure 2: Intermodule calling relationships



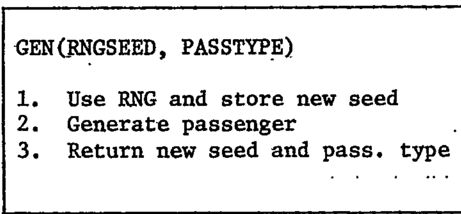
In addition, the importance of indirect relationships through modification of common data areas was not recognized. Many FORTRAN based simulation models use a large area of COMMON to store system state variables which can be modified by numerous subprograms. This makes debugging very difficult. Parameterization is to be preferred if at all possible as it makes correction and enhancement easier and safer. Figure 3 illustrates a program module that generates a potential passenger and places him in the appropriate queue according to his type.

Figure 3: Poor Module Design



This module can be criticized on a number of grounds. Its direct modification of common data (QUEUE) gives it an indirect influence on an unknown number of other modules, and makes simple changes to the QUEUE or RNGSEED data structures unnecessarily dangerous. Furthermore its mixing of two functions, using the Random Number Generator (RNG) and placing a passenger in a queue, makes the module's functions unclear. Figure 4 shows an alternative, and better, modular design.

Figure 4: Good Module Design



Here the module has no direct access to the data areas and the module interface is clearly defined. Books by Myers [11] and Yourdon and Constantine [12] give a lot of useful guidance on successful modularization.

Stepwise refinement is frequently used in developing simulation programs. It allows a high level description of a model, often written in pseudo-code, to be refined down to the level of a simulation or general purpose language. Figure 5 shows the overall structure of a bank "zip line" simulation.

Figure 5: Pseudo-code of Model

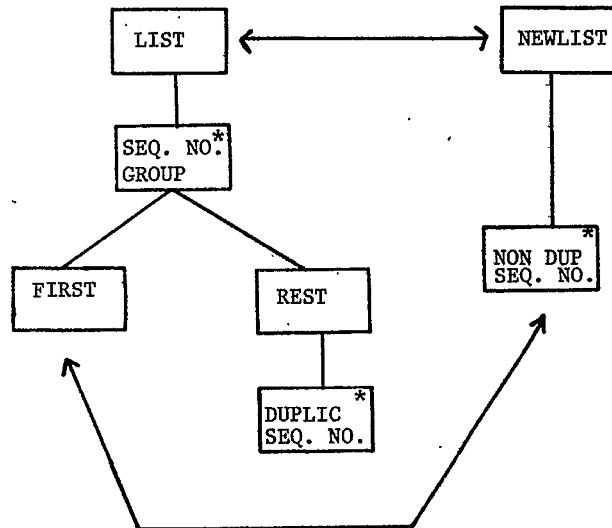
```
initialize
repeat
     find time and type of next event
     case (type of event)
          : arrival of customer
            if queue is empty and some
            teller idle
            then
               serve this customer
            else
               put in queue
            end if
          : completion of service
            if queue is not empty
            then
               serve next customer
            else
               mark teller idle
            End if
     end case
until no more customers
print results
```

In the final or 'target' language, the pseudo code can be retained as comments, giving an overall guide to the program structure. The method of stepwise refinement is well described by Wirth [13]

An alternative top-down approach is Structured Design. This is based on concepts of data transformation and provides a set of techniques which can assist in design. It is described by Yourdon and Constantine [12] and incorporates many ideas of Myers [11] on modularization as described earlier. On the other hand its guidance for developing program structure is limited. In large programs and simulation programs are often very large, it is the choice of overall structure that is most difficult.

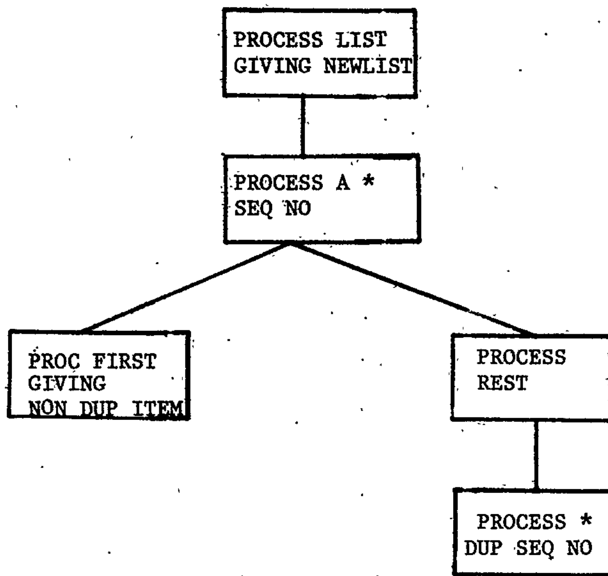
A number of more rigorous methods are available for devising specific algorithms. The "data structure" method of Jackson [14] and the similar method of Warnier [15] are among the most widely used. This approach is most suited to file processing problems as it works on the premise that input and output data (file) structures determine the program structure. In simulation modelling this method can be used when processing inputs (e.g. checking parameters) or when producing outputs (e.g. report generation). In some cases it works very well with internal data structures. As a simple example consider the problem of copying a linked list of items that are held in ascending order of sequence number. Where more than one item has a given number only the first is to be copied. In Jackson notation the data structures, input and output are as shown in Figure 6.

Figure 6: Input/Output Structure



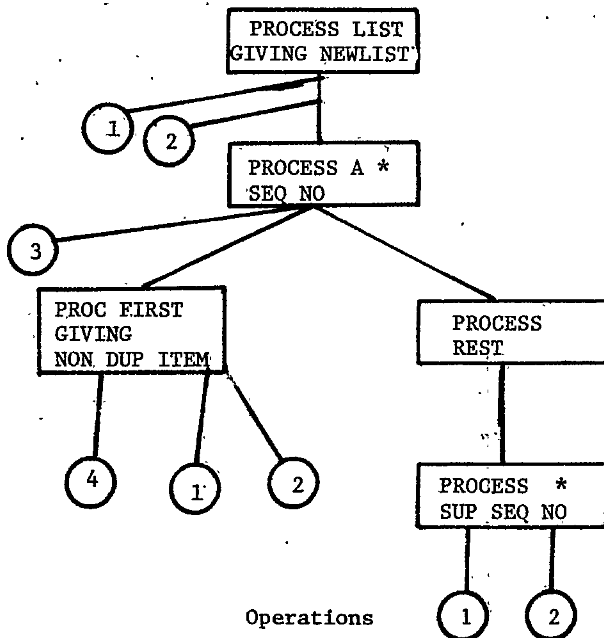
Having established the necessary correspondences shown by the arrows, the overall program structure is obtained as in Figure 7.

Figure 7: Program Structure



Basic operations are then enumerated and attached to this structure, as shown in Figure 8.

Figure 8 Allocated Operations



1. Get item from LIST
2. Extract item sequence No.
3. Store current sequence No.
4. Add an item to NEWLIST

Transcribing this program structure yields the pseudo code result shown in Figrue 9.

Figure 9. Pseudo-code of example program.

```
Begin   PLIST
        Get item from LIST
        Extract item sequence No.
        Repeat while not end of LIST
               Store surrent sequence No.
               Begin  Process first
                      Add an item to NEWLIST
                      Get an item from LIST
                      Extract item sequence No.
               End Process first
               Begin Process rest
                      Repeat while stored no =
                             extract no and not
                             end of LIST
                             Get an item from LIST
                             Extract item sequence
                             no.
                      end  Repeat
               end Process rest
        end    Repeat
end  PLIST
```

Program Verification

The greatest aid to program verification is corrert program design. Next in importance are clarity, style, and ease of understanding. Very often simulation models are poorly documented, especially at the coding level. Code quality can be improved by adhering to the conventions of structured coding. These ensure that all programs are formed from three basic constructs, shown in Figure 10, which can be combined and nested to an abitary depth.

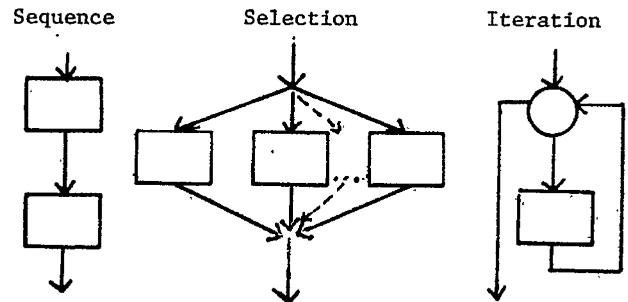


Figure 10 Basic Structures

The purpose and use of these conventions are described fully in Chapter 4 of Jensen and Tonies [15]. Code reading should also be encouraged, particularly for novice programmers and those whose only experience has been in FORTRAN, BASIC or less structured languages. Group sessions to review program design and coding may detect errors and are also useful in encouraging good documentation to be produced. If desired these sessions can be formalized as 'structured walk-throughs'.

Establishing program correctness forms part of the verification stage in the development of a simulation model. Not all techniques are equally useful, however. Program proving is a desirable goal but is impracticable for most simulation programs. However, we can make use, informally, of loop invariants to ensure proper termination of loops. Expressions such as: "Until there are no more arrivals, or queue is full or simulation period is over" will help to ensure consideration of all valid cases. Testing should be looked upon as the last resort. Its purpose is mainly to confirm that the program works. It cannot prove the program correct, least of all a simulation program which will often result in millions of machine instructions being executed. If trace information must be used, it should be judiciously chosen to give the most relevant information. Otherwise the user is swamped in printout. Individual languages have helpful facilities in this regard, for example the PL/C [16] language allows statements (e.g. for output) that can be treated as comments for some compilations but can be taken as source code by changing only compiler options. Such statements are called compilable comments.

System Optimization

Simulation programs are big users of computer time and storage. When we try to economise, however, we should try to concentrate our efforts where they will have most effect. With regard to storage use, we can often estimate the amount of array or table storage as a function of the simulation parameters. For example, in a bus route simulation (Ryan [17]) the maximum storage requirement was found to be

$$S = 20n_s + 26n_r + 8n_b + 82n_s n_r + 10n_s n_t + 8n_b n_{jt} + 4n_s n_{ar} + 4n_s n_{jt} + 458$$

where: n_s is the number of bus stops in the system

n_r is the number of bus routes

n_b is the number of branch points

n_t is the number of passenger types

n_{jt} is the number of time periods in the journey time profiles

n_{ar} is the number of time periods in the arrival rate profiles

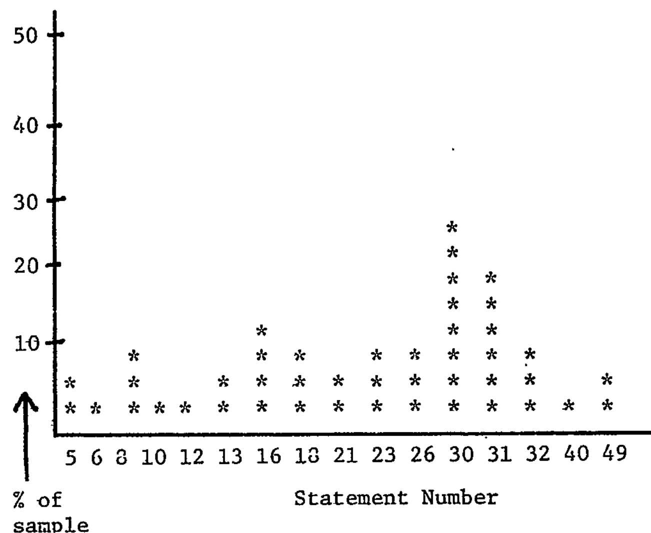


Figure 11 Typical PLITRACE Output

This type of expression allows us to judge the sensitivity of storage needs to variations in the model's major parameters. In this particular case the number of bus stops (n_s) is clearly the most influential on storage needs. Code space is more difficult to estimate. In a statically allocating system, overlaying techniques can be used to advantage, although limited by the main event loop contained in most models. Dynamic allocation, especially with virtual memory is very difficult to optimize and the modeller is usually at the mercy of the page replacement algorithm.

The most frequently quoted performance statistic is the model/real time ratio. One approach to improving this figure is to use a software monitor to provide a relative cost historgram for the program statements. The statement count facility provided in many language compliers is an inadequate tool for this purpose as it reflects only the <u>number of times</u> a statement was executed and not the <u>proportional cost in CPU usage</u> of the statements. For example in the bus route model written in PL/1 and mentioned above [17], a software monitor PLITRACE [18] was used. This operates by periodically interrupting the simulation program and accumulating statistics on the relative frequency with which each statement was interrupted. Its output is in the form shown in Figure 11, a histogram relating statement numbers and frequency of interruption. When applied to a 281 statement simulation procedure. PLITRACE gave the surprising result that 5 statements accounted for over 65% of the CPU time. The 'worst' statement took over 32% of the total. It was coded as:

```
IF EARLIEST >= STIME &¬ALL(BUSES.INUSE) THEN DO
```

Under the PL/1 F-level compiler <u>both</u> of the subclauses would be evaluated before applying the 'AND' operation. In practice, the first clause (EARLIEST >=STIME) would be false most of the time, so that the invocation of the built-in

function ALL would be redundant. As built-in functions appeared expensive in CPU time, (two more of the five worst statements contained the SUM function) it was better to avoid unnecessary use of the ALL function. The statement was recorded as:

```
IF EARLIEST¬=STIME THEN
    IF  ALL(BUSES.INUSE) THEN DO;
```

This single change gave a 24% reduction in the CPU time used. Not all models can be improved so dramatically, but it is generally true that premature optimization is always bad for program structure and is frequently misdirected.

BIBLIOGRAPHY

1. Dijkstra, E.W., "Cooperating sequential processes," Technological University, Eindhoven, The Netherlands, 1965. (Reprinted in Programming Languages, F.Genuys, ed., Academic Press, 1968).
2. Brinch-Hanses, P. Operating Systems Principles Prentice Hall, 1973
3. Hoare, C.A.R. "Monitors: An Operating System Structuring Concept" Comm.A.C.M. 17,10 (Oct, 1974)
4. Shaw A. The Logical Design of Operating Systems, Prentice-Hall, 1974
5. Coffman, Jr., E.G., Elphick, M., and Shoshani, A. [1971]. "System Deadlocks," ACM Comput. Surveys 3, No.2 (June), 67-68
6. Holt, R.C. [1972]. "Some Deadlock Properties of Computer Systems," ACM Comput. Surveys 4, No. 3 (Sept.), 179-195.
7. Mihram, A.G., Simulation: Statistical Foundations and Methodology. Academic Press, 1979
8. Blanning, Robert W. "The Construction and Implementation of Metamodels". Simulation, (June, 1975)
9. Gilb, T., Reliable EDP Application Design, Petrocelli Books, 1974
10. Gilb, T. and Weinberg, G.M., Humanized Input, Winthrop, 1977
11. Myers, G.J., Reliable Software through Composite Design, Petrocelli/Charter, 1975
12. Yourdon, E. and Constantine, L.L., Structured Design, Prentice Hall, 1979
13. Wirth, N., Systematic Programming: An Introduction, Prentice Hall, 1973
14. Jackson, M.A., Principles of Program Design, Academic Press, 1975
15. Jensen, R.W. and Tonies, C.C., Software Engineering, Prentice Hall, 1979
16. Conway, R.and Gries, D., Introduction to Programming-A Structured Approach Using PL/C., Winthrop: 1974
17. Ryan, K.T., The Development and Evaluation of Simulation Models for Bus Operations Planning, Ph.D. Thesis, University of Dublin, Trinity College, August 1977
18. Abrahamson, D.M., PLITRACE, Trinity College, Dublin, Internal Report, 1976