A simulation experiment using two languages

R. H. Perrott*, A. K. Raja* and P. C. O'Kane†

An experiment to test an operating systems simulation language SIMONE (based on PASCAL) on a more normal simulation problem is described. The actual problem chosen was the Radiology Department of a hospital, a model of which had already been constructed in FORTRAN. This enabled not only an evaluation of SIMONE to be performed but also a comparison with the FORTRAN model to be undertaken.

The comparison data showed a substantial reduction in the timings for the SIMONE model. The advantages of using a language with appropriate simulation features, richer data and control structures, good debugging and tracing facilities were also evident.

(Received April 1978)

1. Introduction

This paper reports on a project which concerns FORTRAN and the PASCAL-based language SIMONE (Kaubisch, Perrott and Hoare, 1976); the latter was originally developed to simulate operating systems algorithms.

The extensions which were introduced to PASCAL (Wirth, 1971) enable a user to experiment with situations which involve parallel actions, viz processes, monitors and condition variables. Processes are those parts of a program which can be executed in parallel; monitors (Brinch Hansen, 1973; Hoare, 1974) are used to control the interaction between processes and condition variables (available within monitors only) are used for process synchronisation by means of 'wait' and 'signal' operations.

The designers of SIMONE felt that the language had more widespread applicability and so it was decided to apply it to a more normal simulation problem. Fortunately, one of the authors (O'Kane, 1976) had collected data and had constructed a simulation model (in FORTRAN) of the radiology department of a local hospital. Hence, data for a more normal simulation problem already existed and a model was also available for comparison purposes. The simulation model of the radiology department was, therefore, rewritten in SIMONE so as to test SIMONE's features and also to compare the two languages. The original logic of the FORTRAN model was followed as closely as possible in order to facilitate the comparison. SIMONE was not found to be lacking in any features and, in fact, was found to be very suitable for the construction of such a simulation problem.

When the compilation and execution times were compared, the difference in SIMONE's favour was found to be great; even taking into account the difference due to the FORTRAN implementation. This clearly emphasised the need for using a language which has features appropriate for simulation. The clarity of the SIMONE model and its ease of construction can be attributed to the richer data structures and structured programming concepts inherited from the PASCAL language. In addition, the compile time and run time checking, and tracing facilities of SIMONE reduced the effort required for program debugging.

The next sections describe the radiology department, the language SIMONE (briefly), the SIMONE model and the comparison of the two programs.

2. Problem

This section contains a description of the main characteristics of the radiology department which was under investigation, and identifies the parameters which are important for the simulation model.

1. *The radiographers*

A given number of radiographers was available for carrying out examinations. In addition to the actual film taking, there were a number of checking and clerical duties which radiographers were expected to carry out in relation to each patient.

Although the department provided a 24-hour service, the main organisational problems occurred during the 'working day' which has been defined as extending from 9.00 a.m. to 5.00 p.m. During this period, each radiographer is permitted three rest periods, the duration and times of which are specified but the latter could be varied if the work load demanded. The radiographers worked a five day week.

2. *The examination rooms*

Five rooms were available for examinations. These were not all equipped in the same way and, thus, any given type of examination could be performed in a subset of the rooms only. When a patient's examination was completed in a given room, a certain time is necessary for preparation before another patient is admitted to that room.

3. *The examinations*

There were 22 different types of examination and an empirical distribution was used to find the time required for a particular examination. The data indicated that relatively few patients had the same examination performed more than twice on a single visit, or had combinations of more than two types of examination. Hence, this information was incorporated into the model.

As mentioned above, a particular type of examination could only be performed in certain of the rooms; a priority ordering of the rooms for each examination was, therefore, established beforehand.

4. *The patients*

Patients arrived in the radiology department from the casualty department, the wards, a number of different outpatient clinics and by appointment. A negative exponential inter-arrival distribution was used for all of these sources except the last for which predetermined times were available.

The patients on arrival were classified into one of eight sources depending on the particular department or clinic from which they originated (there were five clinics in the outpatients department). Hence, the pattern of arrivals varied for each

*Department of Computer Science, Queen's University, Belfast BT7 1NN, Northern Ireland
†Department of Business Studies, Queen's University, Belfast BT7 1NN, Northern Ireland

source, from day to day, and even within a given source; a source could, therefore, be active for only part of a day.

A single queue was formed by the patients from the eight sources as they waited for a suitable room and a radiographer to become available. An attempt was made to maintain a first come first served discipline; however, a patient was passed over in a search pass if the facilities it required were not available immediately.

3. The SIMONE language

SIMONE is a discrete event simulation language which is based on PASCAL (Wirth, 1971). The language was originally developed to provide a teaching aid to support a course in operating systems for students who were already familiar with PASCAL; it soon appeared that the language had more general applicability. This experiment is an attempt to use the language on a problem which does not have the characteristics of an operating system simulation, i.e. in an environment where the identity of a process is important.

The language provides a quasiparallel programming facility like that embodied in the process and class concepts of SIMULA 67 (Birtwistle *et al.*, 1973). The actual compilation, testing and simulation (for reasons of efficiency) are carried out in a normal fast batch environment.

Three basic features were introduced into PASCAL to enable a user to experiment with quasiparallel programming, viz. processes, monitors and condition variables. Processes are those parts of a program which a user wishes to represent parallel activity; each process can be imagined as an independent sequential PASCAL program. All the processes progress independently with respect to a simulated integer timescale which is increased automatically when all activity at a particular time has been completed.

Monitors have been provided to control the interaction and communication among the processes. A monitor consists of a set of shared variables, together with their initialisation, and the procedures or functions which manipulate them. In an operating systems environment, the monitors are used to isolate critical regions from the processes.

A process can access the variables of a monitor by calling one of the monitor's procedures or functions; simultaneous updating of the shared variables is avoided by allowing only one process at a time to enter a monitor; mutual exclusion is thus guaranteed.

The third main feature is condition variables which enable processes to co-operate and synchronise; they are used only within monitors. When a process enters a monitor, it can suspend itself on a condition variable pending the action of another process. Hence, a condition variable identifies an ordered queue of suspended processes. The queue may only be manipulated by two operators 'wait' and 'signal'; the 'wait' operator causes the process which invoked it to be attached to the queue, while the 'signal' operator detaches (and causes immediate resumption of) the process with the highest priority. The 'wait' operator may have an integer priority value associated with it to represent the importance of the process. If the queue is empty and a signal operation is performed, the effect is null.

The method used to manage store in the SIMONE language is simple; it makes the naïve assumption that all the processes will be active simultaneously; hence, the determination of the space required is static. A logical outcome of this decision is that space recovery is unnecessary (since enough space has been allocated for all the processes to run simultaneously, the space will never be reused). Hence dynamic creation of processes is not possible.

However, if there is a large number of processes in any program such that the available store is insufficient to accommodate them, the user must set up the recycling of process space. This was found to be necessary in the current experiment. A more sophisticated store management algorithm, such as scan mark garbage collection would avoid this problem.

Several other features enable the user to effect closer control over the simulation under investigation. In particular, a system integer variable 'time' contains the current value of simulated time. The passage of time is achieved by using the standard procedure 'hold', e.g. a process may represent some activity by calling 'hold(*n*)' where the integer parameter represents '*n*' units of time. Full details of the language are given in Kaubisch, Perrott and Hoare (1976).

4. The SIMONE model

The model can operate for any number of working days and consists of a number of processes interacting by means of monitors. A main process is responsible for controlling the daily cycle by initialising the conditions for each day. It also maintains the information on the source arrivals.

1. *The patients*

Each patient is represented by a process. The patients with appointments on a particular day are first scheduled and then the other patients are generated as dictated by the distribution, i.e. their arrival times, types of examination and corresponding times for examination are determined. As the day progresses, the processes are appropriately generated to have their examination.

If the system is simulated for many days, the number of patients can run into several thousand. Since each process (representing a patient) requires 45 words of store and store is not recovered by the SIMONE system (see last section), a

```
monitor patientgeneration;
var cycleq: condition; /* a queue of available processes */
  /*other data declarations */
  procedure suspend(/* distribution parameters */);
  begin
    cycleq.wait;
    /* accept parameters of the generated arrival */
  end suspend;
  procedure activate(/* distribution parameters */);
  begin
    /* assign parameters of the generated arrival */
    cycleq.signal /* release a patient into the system */
  end activate;
begin
  /* initialisation of monitor variables */
end patientgeneration;
process patient;
begin
  while true do
  begin
    patientgeneration.suspend; /* patients wait for activation */
    /* enter the Radiology Department for examination */
  end
end patient;
```

Each time an arrival is due, the main process generates a call

patientgeneration.activate(/* actual parameters */);

thus passing the parameters to the process released from the 'cycleq' queue in the monitor. After a process has completed its examination in the radiology department, it rejoins the 'cycleq' queue—later, it can assume the characteristics of a different patient (if necessary).

Fig. 1 Generation of patients

```
procedure seekservice;
var found : boolean; patientqueue; condition;
  serialno, patientno : integer;
begin
  serialno := serialno + 1;
  patientno := serialno; /* patient's priority */
  if neither a suitable room or radiographer is available then
  repeat
    repeat
      patientqueue.wait(patientno);
        /* patients wait in the order of their arrival */
      /* is a radiographer available */
    until radiographer available;
    found := false;
    if is room suitable
      then found := true
      else patientqueue.signal /* release next patient in the queue */
  until found;
  room := busy;
  radiographer := busy
end seekservice;
```

The monitor procedure 'seekservice' is called by a patient after entering the Radiology Department.

Fig. 2

```
procedure releaseroom;
begin
  room := free; patientqueue.signal
end releaseroom;
```

This procedure is called when a room becomes available.

Fig. 3

```
procedure freerad(var period : integer ; restflag : boolean);
begin
  radiographer := free;
  if (all rest periods not utilised) and (rest period soon) then
  begin
    restflag := true ; period := restduration;
    radiographer := onset
  end else
  begin
    restflag := false ; room := 1;
    while (patientqueue ≠ empty) and (room <= 5) do
    begin
      if room not busy then patientqueue.signal;
      room := room + 1
    end
  end
end freerad;
```

The above procedure is called when a radiographer finishes the examination of a patient.

Fig. 4

severe store shortage could exist. To avoid this problem, an estimate of the maximum number of processes that could be active at any time was made; these processes were then reused or recycled as required. This was achieved by means of a monitor and a condition variable, as illustrated in **Fig. 1** (only sufficient variables and comments are given to enable an understanding of the technique and the features of the language).

When a patient enters the system, it is assigned a number which represents its priority while it is in the system. The priorities are increasing integer values with the lowest number always representing the highest priority. In order to be examined, a patient consults a monitor to see if a suitable room and a radiographer are available. If either or both of these requirements is not satisfied, the patient joins a waiting queue; its position in the queue is determined by its priority (**Fig. 2**). Whenever a room (**Fig. 3**) or a radiographer (**Fig. 4**) becomes available, it causes the queue to be searched to find a suitable patient; this involves removing a patient from the queue and examining its parameters; if unsuitable it is reinserted at the same place in the queue. This is continued until a suitable patient is found or the complete queue has been searched.

2. *The radiographers*

A process was used to represent each of the radiographers. The three rest periods caused some difficulty in organising the radiographers' activities since there was some flexibility as to when a rest period could commence.

1. A radiographer would not start an examination if it would cause an excessive delay in the start of the next rest period—an excessive delay was arbitrarily defined as twice the duration of the rest period.
2. If there were no patients waiting and the next rest period was due within five minutes, it could start immediately.

In fact, another process 'restclock' was associated with each of the radiographers in order to give the alarm when a rest period became due.

3. *Other processes*

Another process type was required to keep a room 'locked' for the period when it was being prepared for the next examination.

5. Comparison

The radiology department was simulated for 125 days with four radiographers working and five rooms available for examinations. The programs were executed on an ICL 1906S and the results are shown in **Table 1.** The timings for the compilation of the programs are those given by the operating system and include all operating system overheads; however, the execution time of each program was monitored by the program itself.

The original FORTRAN (A) program was used, as far as possible, as a guide for the construction of the SIMONE program to facilitate comparison. The basic logic of both models is similar and the output from both models is compatible. FORTRAN (A) passed all the data to its subroutines using parameters; this program was modified by introducing COMMON areas, wherever possible, to improve its performance; this is referred to as FORTRAN (B). This caused only a slight improvement in performance, and reduction in

Table 1

	SIMONE	*FORTRAN* Original model (A)	Modified with COMMON areas (B)
1. *Compilation*			
Time (seconds)	14·1	23·4	24·4
Store used	28K	31K	31K
Disc transfers	65	1107	1115
2. *Execution*			
Time (seconds)	51·01	290·93	279·28
Store used	12·4K	18K	17·5K
3. Number of lines of code	672	1090	1183

A comparison of either FORTRAN program with the SIMONE program shows a remarkable improvement in compilation time, transfers, execution time and storage used. This is a much greater all round improvement in performance than expected and some of the reasons for this will now be offered.

the storage required. (Any improvement is a consequence of using 1900 FORTRAN.)

1. *Compilation*
SIMONE is a one-pass compiler and, therefore, faster than the multipass FORTRAN compiler. The multipasses in FORTRAN are also responsible for the large number of disc transfers used.

2. *Subroutine/procedure calls*
In SIMONE, the data areas are allocated dynamically on each procedure call requiring extra instructions to be executed. In the case of monitor procedure calls, additional instructions are required. The example under study makes heavy use of procedure calls (estimated at over 126000); hence the overhead which results constitutes a substantial part of the SIMONE execution time.

In the FORTRAN programs, there is no run time administration of the data areas required because all storage is allocated statically. Hence subroutine calls do not contribute a significant overhead.

3. *Queue manipulation*
In SIMONE, a patient can be delayed on one of three queues as follows

(*a*) the sequencing stack—processes waiting to execute at the current instant of simulated time
(*b*) the time queue—processes arranged in order of restart time
(*c*) a condition variable queue, with processes arranged in order of their priority.

The queues are maintained and manipulated by the built-in routines (machine language) in the run time system. Hence, the manipulation of the queues is achieved in an economical number of machine instructions.

In the FORTRAN programs, the user has to set up the appending and removing of patients from a single queue explicitly in FORTRAN, and this is a contributing factor to the higher execution time.

4. *Event selection*
In SIMONE, the patients are arranged in chronological order, i.e. in the sequence in which they resume execution, by the system. The run time routines are also responsible for the passing of control from one process to another whenever a change of state occurs. Hence, the overhead for selection of the 'next event' is very small and handled automatically by the run time system.

On the other hand, the incorporation of a means for selecting the 'next event' for execution in the FORTRAN programs introduced many complexities, and is a major contributing factor to the higher execution time.

5. *Storage*
The richer data structures of SIMONE were appropriate for the representation of the composite data types which occurred in the problem and led to a saving of storage. For example, the data used to generate the time required for an examination consisted of a pair of numbers; a real number to indicate the probability and an integer to represent the time. In SIMONE, this can easily be represented by a two field 'record', one field being of type integer and the other real.

In the FORTRAN programs, this was represented by a two dimensional array of type real with its corresponding storage overhead. In fact, the FORTRAN programs made heavy use of arrays of several dimensions (and the extra storage involved) because a more flexible data structure, such as the record, was unavailable. The enumerated data type of SIMONE also enabled a more meaningful and readable representation of many of the features in the model.

6. *Tracing/debugging*
SIMONE has, in addition to the normal PASCAL compile and run time checking facilities, automatic tracing (Perrott and Raja, 1977) and statistic collecting features which help the user to understand and to debug his program; these can be turned on or off selectively but still contribute a time and space overhead in either case.

In the FORTRAN model, there was no such overhead. In fact, statements had to be inserted throughout the program to catch such information; those statements took time to devise, increased the size of the program, obscured its logic and increased the debugging effort. However, the figures reported have all such additional statements removed.

Hence, the difference in the execution timings can be attributed to a combination of the above effects, in particular the lack of queue manipulation and event selection features in the FORTRAN model; however, we still did not expect that the difference would be as great as a factor of five in the execution times.

There were other (less quantitative, but important) benefits of using a language such as SIMONE. The rich data structures, programming and simulation concepts enabled the model to be designed and implemented in a systematic and structured manner. This higher level of abstraction enabled the construction of the model in a shorter period of time, producing a shorter program and thus reducing the chance of making errors. This also increased the program's clarity, readability and ease of understanding. The availability of compile and run time checks plus tracing facilities reduced the debugging effort; the statistics features made it easier to obtain the required results.

6. Conclusion
The original objective was to test the language SIMONE, which had been specifically designed for the simulation of operating systems, on a more normal simulation problem. The main concern was that the need to obtain the identity of a process, which is not so important in an operating system environment and therefore not so easily obtained in SIMONE would complicate the design and construction of a normal simulation where it is frequently required. Our experiment with SIMONE in this different environment indicates that the anonymity of processes is not a major drawback of the language.

The comparison of the languages would be more accurate if the simulation was performed in FORTRAN and PASCAL (or SIMONE and GPSS). In such a situation, it is unlikely that FORTRAN would fair so badly; however, there is a considerable gap in the execution timings to be reduced. It must also be appreciated that the emphasis of the two studies were different and that they were performed by different programmers. The original objective was to obtain a working model without regard to efficiency considerations. The comparison reported here does, however, emphasise that for simulation purposes, it is important to choose a language that has features which are appropriate to the problem being solved.

The experiment also shows that substantial benefits can be derived from using a language, such as SIMONE, which has relevant simulation features, rich data and control structures and good debugging facilities. A program can be designed and written in less time, with less effort, requiring less coding, resulting in less opportunity to make errors and, therefore, requiring less time to find, diagnose and correct them. The resulting model is also shorter and more readable.

References

BIRTWISTLE, G., DAHL, O-J., MYHRHAUG, B. and NYGAARD, K. (1973). *Simula Begin*, Student Litteratur, Auerbach.

BRINCH HANSEN, P. (1973). *Operating Systems Principles*, Prentice Hall, Englewood Cliffs, NJ.

HOARE, C. A. R. (1974). Monitors: an operating system structuring concept, *CACM*, Vol. 17 No. 10, pp. 549-557.

KAUBISCH, W. H., PERROTT, R. H. and HOARE, C. A. R. (1976). Quasiparallel Programming, *Software—Practice & Experience*, Vol. 6, pp. 341-356.

O'KANE, P. C. (1976). The Operation of a Hospital Paramedical Department—A Quantitative Study, Ph.D. Thesis, The Queen's University of Belfast, N. Ireland.

PERROTT, R. H. and RAJA, A. K. (1977). Quasiparallel Tracing, *Software—Practice & Experience*, Vol. 7, pp. 483-492.

WIRTH, N. (1971). The Programming Language PASCAL, *Acta Informatica*, Vol. 1, pp. 35-63.

Book reviews

Workshop on Reliable Software: Applied Computer Science by Peter Raulefs, 1979, 281 pages. *Carl Hanser Verlag*

This book presents the proceedings of the workshop on reliable software organised by the German Chapter of the ACM and held at Bonn University in September 1978. Twenty-two papers were selected for presentation and inclusion in the proceedings. Ten of these papers were submitted by researchers from Germany, six from the USA, and the rest from Italy, Austria, France and the UK. All but three of the papers appear in English in the book; for the sake of completeness, it is a pity that the remaining three were not also translated from German.

The general standard of presentation of individual papers is good but there is a paucity of introductory material, both to put the overall purpose of the workshop into perspective and to give some background on the major themes covered. There are no reports of panel discussions and audience responses so some of the possible benefits of publishing the proceedings of a workshop seem lost. However, the book is still of considerable interest because it contains a broad range of material devoted to several different aspects of the very important topic of software reliability. Papers are presented on program testing and fault detection; systems for program development, software design methodologies for industry, programming language concepts, specification and documentation, and program verification. Thus, there are papers with a theoretical emphasis—verification of while-programs in a simple calculus, for instance—and others with a more practical bent such as the application of new testing techniques to a numerical algorithms library. The book should be of interest to both the more theoretically inclined and to those with a practical concern in the production of reliable software.

BRIAN FORD (Oxford)

Reliable Software Through Composite Design by G. J. Myers; 1975; 159 pages. (*Van Nostrand Reinhold*, £5·20)

I am not sure why this book has only just come up for review—although the review notice stated that it was published in 1979, the book itself gives a 1975 date—perhaps having been produced in America it is only now being distributed in the UK. It is written by Glenford Myers who is already a well known authority on software production. He is the author of many research papers and a number of books and contributes to some of the 'up-market' seminars in this area.

This particular book of his describes what Myers calls 'composite design'. This is a design methodology which draws on many of the ideas often associated with Constantine, namely the functional decomposition type of approach, to be contrasted with the data structure methods of Jackson and others. Thus the book, after a good preamble in which 'program quality' is defined, deals first with the major attributes of modules—termed 'strength' for the internal relationships within a module, and 'coupling' for the links between modules. Other attributes of modules, such as size and predictability are discussed, and for each recommendations are given on what one should be aiming to achieve for a 'good piece of software'. The actual process of dividing a problem into modules is discussed under the heading of composite analysis using some good examples to illustrate the method. Concluding the part of the book specifically describing 'composite design' is a chapter which shows how it fits in with structured programming, documentation and other aspects of software production. There is then a chapter on modularity and virtual storage which I felt was a 'poor mans' version of his paper in one of the IBM journals. The book ends for me, however, on a very strong note with a 'model of program stability' in which the attributes of a piece of software are first quantified and then predictions can be made about the 'ease of change' of the software. Although some would see this venture as rather academic, the chapter does underline the reasons for aiming at good module design, and serves as a good conclusion and summary of much of the rest of the book.

The book is designed as a practical guide for experienced programmers and system analysts. It gives details of theories from the viewpoint of their use for both application program and system program design. I believe the book succeeds in these objectives: it is clear, easy to read and understand, and is not too long. Myers, almost apologetically, states that 'Composite Design is largely a collection of guidelines', and the book details these 'guidelines'—'not firm rules'. As for reliability, this is discussed briefly in the introduction and then it is assumed that the reader can 'see' that employing the methods of 'composite design' will lead to reliable software.

This is a useful guide for people engaged in the production of software systems on how to design the internal structure of a program.

DAVID JACOBS (Leatherhead)

Fortran, PL/I and the Algols, by Brian Meek, 1978; 291 pages. (Macmillan, £12·00)

This book is not an introductory programming text for the languages in its title, nor is it yet another book on the comparative study of programming languages. It would seem that the book falls nicely between two stools. Indeed that was the reviewer's opinion before reading the book. Brian Meek has in fact identified a completely empty stool and provided the appropriate book.

The book discusses six languages—ALGOL 60 (Revised and Modified), ALGOL 68, FORTRAN IV, FORTRAN 77 and PL/I—from the viewpoint of a user not the viewpoint of a 'language lawyer'. This provides a refreshing change. Indeed the book is very pleasant to read, being marred (for me) only by a galaxy of seemingly irrelevant quotations; although I must admit to liking one quotation from Aldous Huxley. Inevitably there are a few typographical errors, but none of them will cause the reader any problem. They will irritate the author more than the reader. I did not notice any errors in the language details. I do disagree with some remarks about Euclid's Algorithm and Ackermann's Function.

If you are involved with teaching a course on any of the languages or a course on a comparison of languages, or are simply interested in programming languages you should definitely read this book. The book will also make excellent background reading for a student on either course, but its choice of languages and its price suggest that it is not suitable for use as the text book for a course on the comparison of languages.

Regrettably the publishers chose to produce this book only in hardback; a paperback edition would surely have been at a price more attractive to students.

A. M. ADDYMAN (Manchester)