Systems with state re-set

J. J. Florentin* and A. J. Sammes†

Some computer systems, such as operating systems and communications controllers, have to work continuously for long periods. These systems are almost inevitably affected by intermittent errors, and there is a need to develop design methods which minimise the effect of occasional errors. This paper points out that, if some part of a system can be modelled as a finite state machine, then it is sometimes possible to arrange that this part will reset itself to a correct state after an error. The technique has been used in the design of a simple interactive programming system.

(Received September 1973)

1. Introduction

Many computer systems have to work continuously for long periods; typical systems of this kind are machine operating systems and communications controllers. It is virtually inevitable that some hardware and software malfunctions will occur, and it is necessary to provide mechanisms and procedures for system recovery after such errors. Some errors are minor, and do not lead to major reconstruction of corrupted data; all that is needed is to reset a program, or logic device, to a correct operating state. If the action of the program, or device, can be modelled as a finite state machine then it is sometimes possible to arrange that this resetting will take place automatically after a few faulty steps of operation; this may well be an adequate recovery if errors occur infrequently. This paper explains how a finite state machine can be made to reset to a standard state by a sequence of input commands, and briefly describes how the control program of a simple interactive programming system based on a model, was examined for self-correcting behaviour against intermittent errors.

It can be expensive and tedious to track down the causes of infrequent minor errors, and when the causes are found it may not be practicable for the system designer to remove them. Consider the following ways in which errors might be caused:

(*a*) A program properly meets its specifications, but the application problem was not fully understood, so that the program occasionally produces inapplicable results.

(*b*) The application problem is correctly understood, and a correct program specification produced, but the constructs of the programming language are misunderstood, so that wrong results are occasionally produced.

(*c*) The application problem is correctly understood and a correct program produced, but the compiler is faulty, so incorrect results are occasionally produced.

(*d*) The problem is correctly understood and a correctly compiled program produced, but a hardware malfunction occurs during running, so incorrect results are occasionally obtained.

These examples show that faults can occur at various levels, and a system designer will only have responsibility for a few of these levels. Faults occurring at a level for which the designer has responsibility are *misconstructions*, and in principle can be put right by the designer. Faults at levels outside the designer's control can be called *miscomputations*; the designer can only provide recovery procedures to deal with these. Ideally, a designer should anticipate possible sources of miscomputations and design recovery procedures for them.

2. Finite-state models of parts of computer systems

Computer systems differ widely in their detail and for the purpose of error analysis a simplified model is needed which abstracts the essential features. A well known model for certain parts of computer systems, such as the conditions of a control program, is the *finite-state model.* This is described in texts such as Minsky (1967); a finite number of conditions of the systems are recognised and called *states.* Certain trigger actions in the system are designated as *input commands*, and under the stimulus of an input command the system moves from one state to another. State transitions can be specified by diagrams of the kind shown in **Fig. 1,** where the nodes represent states, and the branches are labelled with input commands.

Experience and intuition is needed to abstract some part of a system to a finite-state model; for example a communications controller might have states 'transmitting', 'receiving' and 'waiting', but it would not be sensible to consider the various conditions of stored data in a database as constituting a finite number of states. Usually only a fragment of a system can be abstracted to a finite state model. This fragment would probably include control modes, but would exclude contents of mass storage. Judgement is also needed in the recognition of the activities constituting an input command.

2.1. *Transition errors in finite-state models*

After a part of a system has been abstracted to a finite state-model its behaviour is simplified to making state transitions. In this theoretical model only two kinds of error can occur, firstly an input command can be corrupted, and secondly the system response can be faulty so that a false state transition occurs; of course both errors can also occur together. In each case the system reaches an incorrect state, and thereafter its states will continue to be incorrect. A first approach to error detection and correction can be made by noting that some command inputs might be invalid when the system is in certain states. A monitoring device could now be added which detects impermissible input/state combinations, and takes corrective actions if they occur. In this approach the appearance of permitted input/state combinations does not guarantee correct system action, since several states may permit the same input; there is thus a state ambiguity. This state ambiguity could be further reduced by considering a sequence of successive input/state combinations; for example, monitoring the last three input/state values. Two drawbacks of this approach are that it is difficult to choose the length of the input/state sequence to be monitored, and that the monitor itself is subject to faults which are not checked. The method given below overcomes both these drawbacks.

*Department of Computer Science, Birkbeck College, London WC1E 7HX.
†Project Management WAVELL, Ministry of Defence, For Halstead, Sevenoaks, Kent.

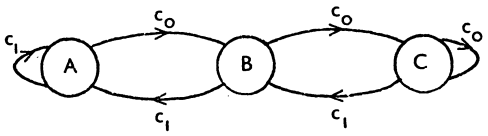

Fig. 1

2.2. *State resetting in finite-state models*

Consider a system specified by the state diagram of Fig. 1, and suppose that a fault has caused the system to move to an unknown, and presumably incorrect, state. Apply the input command sequence 'c_0c_0' then assuming no further errors, the system must be in state C. The input command sequence 'c_0c_0' has restored the system to a known state. Sequences with this effect have been studied in information transmission (see Neumann, 1962), and are called *synchronising sequences*. Not all input sequences are synchronisers; for example, the input sequence 'c_0c_1' leaves an ambiguity over states A and B. Given a state transition diagram its synchronising sequence may be found by constructing a *forward successor tree* diagram. This shows the transitions of whole sets of states under the various possible input commands. The root of this tree is the complete set of states. Each successive node of the tree is found by collecting together the states resulting from each individual input command acting on the states in a previous node. Successive branches of the tree are constructed until a node repeats, or becomes a singleton set. **Fig. 2** shows the forward successor tree for the model of Fig. 1, where each node shows a state set, and the branches are labelled by the input causing the transition.

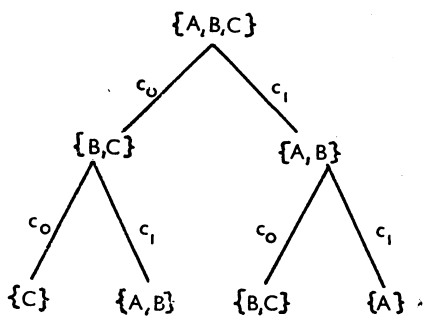

Fig. 2

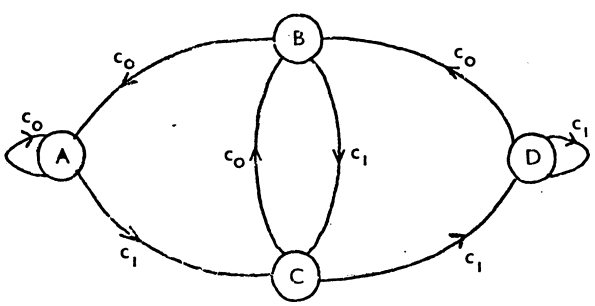

Fig. 3

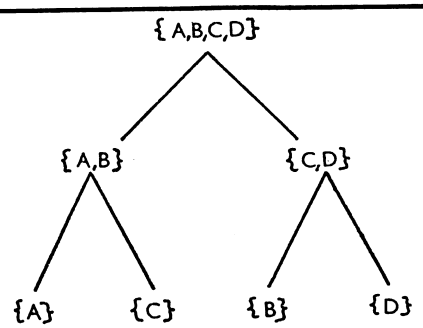

Fig. 4

Fig. 3 shows a four state model, and **Fig. 4** shows its forward successor tree. It will be seen that every input sequence of length two is a synchronising sequence, and the model will be in a known state after any sequence of length two, i.e. 'c_0c_0' leads to A, 'c_0c_1' leads to C, 'c_1c_0' leads to B, and 'c_1c_1' leads to D. A model which is forced to a single state after every sufficiently long input sequence is said to be *fully synchronised.*

A representation of the forward successor tree which can be handled easily in a computer is based on using Boolean transition matrices. Suppose a finite-state model has input command set $\{c_1, c_2, \ldots, c_r\}$, and state set $\{s_1, s_2, \ldots, s_n\}$, then for each input, c_k, there is an $n \times n$ Boolean matrix, C_k, called a *transition matrix*. This has entries d_{ij}, $1 \leqslant i, j \leqslant n$ with

$d_{ij} = 1$ if input c_r drives the model from state i to state j
$= 0$ otherwise.

Further, a subset of states, say $\{s_p, s_q, s_t\}$ is represented by an n-vector with 1's in positions p, q and t, and with 0's elsewhere. Given a state subset (node of the forward successor tree) represented by S, the new state subset, S', caused by input c_k is given by

$$S' = SC_k \ .$$

In practical examples only a small fraction of the possible number of state subsets actually occur in the forward successor tree. The largest system analysed by the authors had 10 states and 21 input commands, but only 44 state subset nodes. The difficulty experienced in going to larger systems arose from presenting the result of the computation, rather than from the computation itself.

2.3. *Theoretical synthesis of fully synchronising finite-state models*

In designing practical systems it is sometimes possible to alter their specification so as to improve their synchronising behaviour; it is therefore interesting to be able to construct fully synchronising theoretical models. One way of doing this is to start with simple resetting models, and then combine them so as to build larger resetting models. The simplest resetting models have as many states as there are different inputs commands, and each input causes the model to go to a particular state. **Fig. 5** shows a two state reset component, and **Fig. 6** shows a three state reset.

The most general inter-connection of elementary reset components is the series-parallel connection illustrated in **Fig. 7.** The boxes labelled $m1$, $m2$, $m3$ represent component models; $m1$ receives the external input as its only input, $m2$ receives a combination of the external input and the state of $m1$ (at the preceding time instant) as input. Going on, $m3$ receives a combination of the external input and the states of $m1$ and $m2$ as input. The series-parallel arrangement has states which are a composite of the states of the component models. Given an input sequence, $i1$, $i2$, $i3$, on receipt of $i1$ component $m1$ will be reset, on receipt of $i2$ component $m2$ will be reset, and so on.

Fig. 8 shows a pair of two state component models which combine to give the model of Fig. 3. The state correspondences are

$$(x, \alpha) \rightarrow A, (x, \beta) \rightarrow B, (y, \alpha) \rightarrow D, (y, \beta) \rightarrow C \ .$$

3. Application of state resetting

The state resetting behaviour of a system is one specialised feature which can be studied after a design meeting the main requirements has been produced. In a favourable design a high proportion of typical input sequences will be synchronising sequences, and the system will have a natural tendency to pull back into synchronism after an error, provided that the time interval between errors is much longer than the duration of the longest synchronising input sequence.

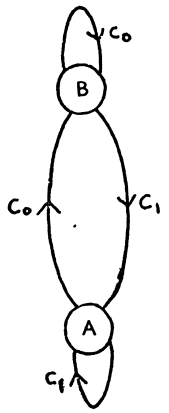

Fig. 5

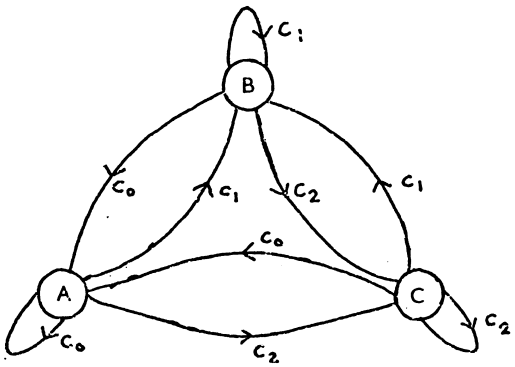

Fig. 6

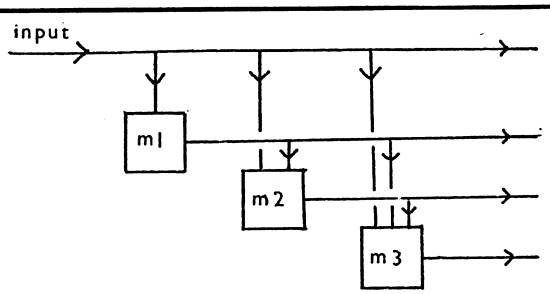

Fig. 7

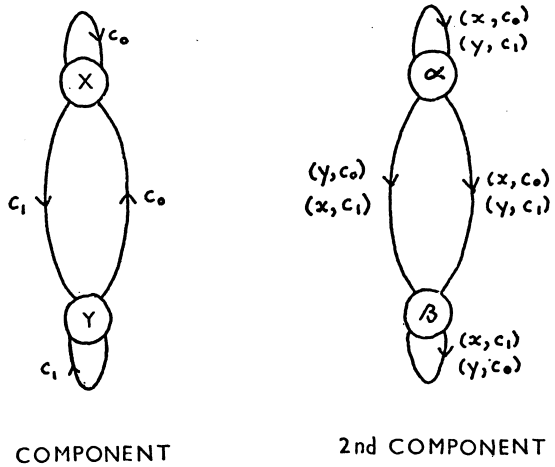

Fig. 8

Studies of synchronising behaviour have been done for the WIPDOS interactive programming system, designed by one of us (A.J.S.) for the Ministry of Defence. This system allows a programmer working at a teletype terminal connected to a small computer, to construct files of program and data, edit these files and then run the programs against the data. The control program was planned around a finite-state model from the outset. Initially, error detection was based on noting certain illegal conditions in the system. Later, the error behaviour was studied by looking for synchronising sequences, some deficiencies were brought to light, and it was possible to modify the design to bring about an improvement.

3.1. *The WIPDOS system*

A finite-state model for the WIPDOS system control program actions was produced before any program modules were written. However for synchronisation analysis the model was simplified still further to reduce the number of states and inputs. The simplified model had 11 states, four of these called EDIT, BASE, USER, and SUSPEND were 'wait' states; the system moved from 'wait' states in response to a command from the programmers terminal. Six states, designated by numbers were 'processing' states, the system performed the requested actions in these states, and moved from them in response to commands generated by system software, or stored files of commands. From the viewpoint of the simplified model (but not the actual processing performed) several commands had the same effect, and were grouped into collections of type A, type B, or type C commands. The transition diagram is shown in **Fig. 9**, and the 21 different types of input command are listed in **Fig. 10**.

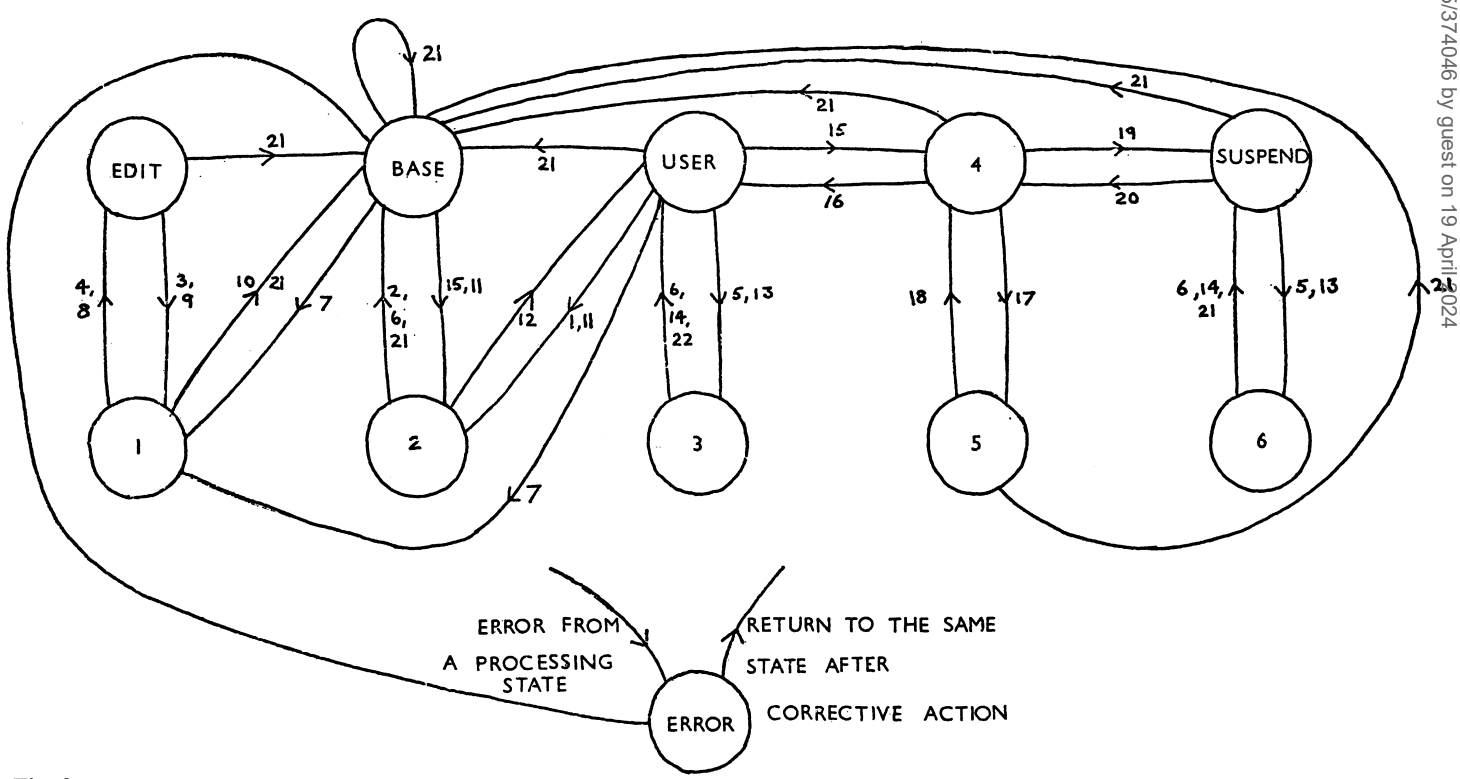

Fig. 9

Input Number	*Command*	*Input Number*	*Command*
1	Type A	12	LOAD return
2	Type A return	13	BREAK
3	Type B	14	BREAK return
4	Type B return	15	RUN
5	Type C	16	RUN return
6	Type C return	17	SVC
7	EDIT	18	SVC return
8	EDIT return	19	SUSPEND
9	ENDEDIT	20	CONTINUE
10	ENDEDIT return	21	KILL
11	LOAD		

Fig. 10

Fig. 9 has been further simplified by showing only transitions from each state caused by permitted commands; impermissible commands cause a return back to the state from which they were issued.

3.2. *Forward successor tree for WIPDOS*

The Boolean transition matrix method described above was programmed so as to print out forward successor trees for any desired command sequence. First the forward successor tree for 'well behaved' input sequences was constructed. 'Well behaved' sequences were those in which only permitted inputs were applied at each state. **Fig. 11** shows the resulting tree where both synchronising, and non-synchronising, input sequences can be found. An example of a synchronising sequence is input 1 (a type A command) which always forces the system to state 2. An example of a non-synchronising sequence is the infinite input sequence 5, 21, 5, 21, . . ., which is a type C command indefinitely alternating with a KILL command.

On the forward successor tree it can be seen that the non-synchronising input sequences include the individual inputs 5, 6, 13, 14 or 21; these inputs are associated with type C, BREAK and KILL commands. When the programmer issues a type C command (input 5) an ambiguity over the wait states 2, 3, and 6. results. Similarly the BREAK command (input 13) leads to an ambiguity over the processing states 3 and 6; the return from the BREAK, input 14, leads to an ambiguity over the wait states USER and SUSPEND. The KILL command (input 21) results in an ambiguity over the wait states BASE, USER and SUSPEND. Further, it can be seen that input sequences constructed only from type C and BREAK commands (and their respective returns) are never synchronising, and always result in ambiguities over, at least, processing states 3 and 6, or the wait states USER and SUSPEND. Sequences made up of type C, BREAK and KILL commands will not be synchronising if the state ambiguity associated with an input immediately prior to the KILL command is over the state sets {2, 3, 6} or {3, 6}.

Summarising these observations—if a miscomputation occurs which causes an incorrect transition to one of the states BASE, USER, SUSPEND, 2, 3, or 6, then the command analyser would remain out of step with the programmers terminal throughout the time that only type C and BREAK commands were being issued. The same effect would occur if the programmer himself lost synchronism by not being aware of the current state of the system. One of the original design aims of WIPDOS was to enable the programmer to readily put the system into a well defined state. After analysing the forward successor tree it was evident that this aim had not been achieved; and that the minimum synchronising sequence associated with the KILL command is 'KILL KILL'. In this particular design it was possible to modify the synchronising behaviour by signalling

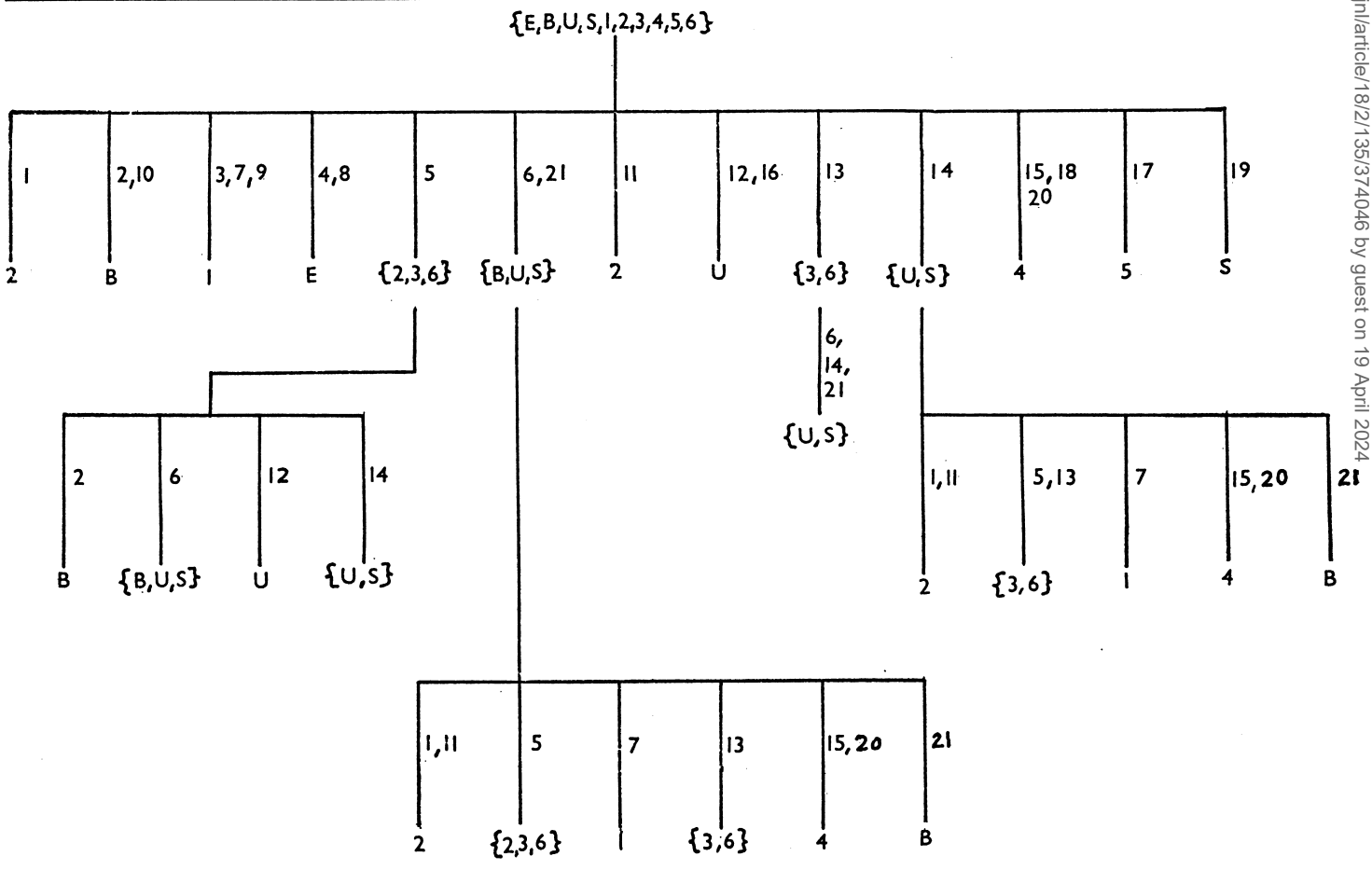

Fig. 11

more information back to the programmers terminal. The modified system then became fully synchronising for well-behaved inputs.

The initial analysis above omitted illegal input sequences; an analysis which included all possible commands applied at every state was later done. A substantially longer computer run was needed, and the printout was much larger. Many of the new sequences were found to be synchronising, but a number of new non-synchronising sequences were also discovered. However, it happened that the system modifications also eliminated most of these new non-synchronising inputs. The overall result of the analysis was a better understanding of system behaviour, and a substantial improvement in the programmer command interface.

Other practical systems have been analysed, and Sammes (1973) describes the analysis of a controller in a computer communications network. In this example an initial analysis was done by computer, but a full analysis was beyond the power of the available computer. An intuitive examination of the tree was done, and uncovered a number of non-synchronising sequences. These were found to explain a number of fault conditions which had been observed experimentally, but were not understood.

5. Conclusions

Examining the synchronising behaviour of parts of systems has proved to be useful. More computer aid would be helpful in this examination. From the authors' experience the most pressing need is for a better means of displaying the essentials of the forward successor tree to the systems designer; an interactive visual display terminal for showing selected parts of the tree is an obvious possibility.

References

MINSKY, M. (1967). *Computation: Finite and Infinite Machines.* Prentice Hall Inc., New Jersey.

NEUMANN, P. G. (1962). On a Class of Efficient Error-limiting variable-length Codes. *IRE Trans. Information Theory*, Vol. IT-8, pp. 260-266.

SAMMES, A. J. (1973). *Error Limiting in Computer Operating Systems.* Ph.D. Thesis. University of London, England.

Book review

Data base management, edited by W. C. House, 1974; 468 pages. (*Petrocelli Books, New York*, £9·95)

This book contains reprints of thirty-five articles and papers originally published between 1966 and 1972, and taken from 23 different periodicals, some relatively obscure. They are arranged in six sections to each of which there is a brief introduction by the editor and to each of which he has appended a page of 'discussion questions' and a bibliography.

The six sections, all about equal length, are entitled: Data collection: minimising time, cost, and error bottlenecks; Data communications: key system components and design considerations; Data organisation and storage; Data base management: an emerging function; Methods of processing data: batch versus continuous processing; and Information retrieval and display: concepts, alternatives and devices.

These headings are more or less a fair description of the material the sections contain and, if one accepts the variability of level of treatment inevitable with a collection of articles and papers originally written independently, they provide interesting reading.

The first section (pages 1–81) contains much discussion on the obsolescence of the punch card for data preparation and the costs and benefits of key to disc system or OCR. Also discussed is the elimination of the data preparation function altogether by making line staff responsible for direct data entry as a by-product of their work. The second section (Pages 85–140) introduces various data communications topics.

The third section (pages 141–226) commences with the 1969 paper by Dodd from *Computing Surveys*, 'Elements of Data Management Systems'. It is followed by several shorter articles with titles 'practical data base design', 'computer storage and memory devices' 'magnetic tape, drum, disc memories', 'selecting computer memory devices', and 'IBM adds virtual storage technology'. The fourth section (pages 227–322) contains mostly general discussion articles with a practical bias and there is some repetition. It includes the 1972 introductory article by Shubert from *Datamation* on the basic ideas of the Codasyl DBTG proposals.

The papers in the fifth section succeed in making the point that batch processing, in some circumstances, is both satisfactory and cost effective but the material tends to range widely and even includes an elementary introductory article, 'Computer Timesharing a primer for the financial executive'. The sixth section (pages 399–468) again ranges widely with discussions on non-impact printers, voice response units, etc.

The reviewer was baffled by the very specific title the volume has acquired. Perhaps it was a mistake! For the blurb on the dust jacket to state that the 35 articles reprinted 'represent the best ideas on data base management' is absurd. The editor's knowledge of this field seems somewhat limited. What we have is some mostly interesting and occasionally useful readings in data processing touching most of the topics in this field. A selection from these articles could well serve as background reading for a general course on data processing. The volume would have been more correctly titled 'Some readings on current themes in data processing'.

P. J. H. KING (London)