Writing simulations in CSL

By J. N. Buxton*

This paper discusses the writing of programs to express simulation problems in the control and simulation language CSL. The descriptions in the text are in terms of a single problem, for which a full program is given as an appendix. Though some new features of the new version of CSL are described, the purpose of the paper is mainly expository.

The word "simulation" is used in many contexts and has become one of the more confusing terms in the computer field. For example, it is used in the simulation of the order code of one computer on another, in the simulation of aircraft or guided weapon control systems by analogue computers, and in operational research. It is the latter use which is implied in the title and for which CSL is designed.

The first version of CSL (Control and Simulation Language) was designed as a joint project by IBM United Kingdom Ltd. and Esso Petroleum Co. Ltd. (Buxton and Laski, 1962; CSL Reference Manual, 1962). Its initial applications were to the solution of problems in operational research connected with the control of industrial organizations. These problems were solved by simulation; that is, by writing in CSL a program purporting to imitate the working industrial system, and then studying the effect of changes in the methods of control of the system by manipulating the imitation program. The basic premise behind this approach is that it is cheaper and less disastrous to experiment with a computer program than to experiment with the real-life system of which that program is an imitation. Furthermore, it may be impossible to experiment on the real system if, for example, it does not yet exist.

During the design period it was realized that a programming language built to imitate real-life systems on a computer is likely also to be suitable for writing computer programs to control the real system. The first major application of CSL, due mainly to Dr. J. G. Laski at Esso Petroleum Co., was in fact a program to control the distribution of petroleum products (Laski, 1965). The control program was written in CSL, tested by surrounding it with a simulation of the distribution system also written in CSL, and subsequently tested on control of the real distribution system. Of the two key words in the title "Control and Simulation Language", the first is probably the more important.

This experiment, and other early uses, revealed many possible areas of improvement which led to the preparation of a second version of the language. The second version of CSL, described in this paper, has been prepared initially for the IBM 7090/94 by IBM United Kingdom Ltd. The compiler has been designed and written mainly by P. Blunden, P. Grant and G. Parncutt of IBM, and the services of the author of this paper were used as consultant in the design of the language. There is little that is basically new in the work; the fundamental ideas of CSL as expressed in Buxton and Laski (1962) still stand, but the language represents a considerable revision and extension. A full technical description is given in CSL Reference Manual (1965) available from IBM United Kingdom Ltd.

Basic system description

The systems for which CSL is used have certain basic properties in common. Their components are described in terms of groups of objects known, for lack of a better word, as classes of entities. All entities in a class are similar in that their basic properties are similar though not necessarily identical. The properties of entities may require quantitative storage or qualitative storage, and the logical interactions between entities may be extremely complex.

This may become clearer after consideration of an example. The specimen program given in the Appendix has as its aim the simulation of the operation of a simple port. The objects handled by a port are traditionally known as ships. All ships are similar, in that they all float, and so we group them into a class of ships. Other classes of entities which may arise are, for example, classes of tugs and berths. This example is, in fact, based on a real industrial problem connected with the handling of ships in a port suffering from insufficient facilities to handle its traffic. Fairly heavy charges are incurred if ships are held waiting at the port before unloading; however, the installation of each extra berth involves heavy capital outlay. The problem of adding the optimum number of extra berths was solved by constructing a simulation model program which imitated the operation of the port, and running this program with varying numbers of berths in the system. The output of each run was statistical data, such as ship queuing times, on the performance of the model for a certain number of berths. These outputs were used to work out costings for a range of possible port configurations.

The isolation by the user of classes of entities is always a little arbitrary; (for instance, for some programs it may be preferable to have a class of tugs; for others it may be more useful to have a single class of floating objects). The basic point is that members of a class are in the main interchangeable.

* *C.E.I.R. Ltd.*, 31, *Newman St., London W.C.*1; and *University of London.*

Numerical operations

The traditional means of holding information describing the properties of objects is by numerical storage. The facilities available for holding this information are similar to those in conventional programming languages; the programmer may define real and integer variables or multi-dimensional arrays of variables, and may perform arithmetic operations on these variables. For example, the loads carried by the ships in the system could be held in a one-dimensional array called LOAD, where the load aboard the X-th ship is held in LOAD (X).

The form of arithmetic expressions and assignment statements is similar to FORTRAN and will not be further described in this paper. Simple test statements, of which there are many examples in the sample program, consist of two arithmetic expressions separated by one of the six relational operators, GE, GT, EQ, LT and LE which have the same meanings as in FORTRAN. The result of success of a test statement is that control proceeds to the next statement. In the event of failure, a statement label can be specified as destination; the meaning of a test failure condition is more fully discussed below.

The handling of statistical information plays an important part in simulation of systems, both as input and output. Storage units called histograms are provided for this purpose; they may be used to accumulate counts of statistical variables, and arithmetic functions exist to take random samples from their populations. Histograms may be grouped into arrays of histograms.

Repetitive operations are carried out by FOR statements, whose properties resemble loop control statements in other languages. The range of repetition of such a statement is denoted by indentation of the group of statements which it controls. Positive or negative increments may be used, and if the range is computed on any repetition to be zero or negative, the group is skipped.

A problem frequently encountered with automatic programming languages in general is that of holding very many items of information, each of which is quite small. This calls for the packing of more than one item of information into each storage word. The facility is provided without difficulty in CSL; an integer array definition may contain a packing factor—specified in the definition—thus:

INTEGER DATA (100,60) PACK 6

The storage items in this array are then packed at six entries per word as signed integers; though the array includes 6000 items, in fact only about 1000 words of storage are used.

Operations on names

The basic premises of the system are that there are two possible ways of holding information about the entities of the system: the quantitative or numerically coded way described above, and a qualitative way which is about to be described. The basis of the qualitative approach is that for many purposes, in particular that of making logical choices, it is preferable to record properties of entities by holding them in groups which share a common property or properties.

This is done by using *entity names.* They are the third type of item which may be manipulated in a CSL program: they differ from the two numerical types in that the entity names valid in a specific program have meaning only in that program. Entity names may be stored in variables or arrays which are declared to be of type NAME, or ordered lists of entity names may be stored in sets. Entity names from any class may be stored freely in any set or name storage; a set is an ordered list of names of indefinite length. Thus, for the purpose of making a choice of a ship to berth in the port, it may be sufficient and more convenient to know whether the ship is large or small rather than knowing in detail its tonnage and load. This can be done by recording the entity names of large ships on a set called LARGE.

There are three main types of expression which can arise in CSL statements; integer, real and name expressions. The system inserts transfer functions freely between the two arithmetic types and thus mixed type arithmetic may be performed. Furthermore, if a name expression is used in an arithmetic context, a transfer function is applied to extract the serial number part of the name and use it as an integer.

In an analogous way to arithmetic statements and tests the language has statement forms for the manipulation of entity names. Clearly arithmetic cannot be done on them, but they can be moved on and off sets, compared with each other, and searched for in sets. Examples of statements carrying out these operations occur in the Appendix.

Sets serve two basic purposes. Their first use is in making logical decisions. For example, in real life the decision to berth a particular ship is based on a selection process which chooses from all ships in the harbour and waiting to be berthed, one satisfying various possible criteria on size, type of cargo, i.e. crated or loose and so on. Now, in the CSL imitation, the user may define sets called INHARBOUR, LARGE, SMALL, CRATED, LOOSE and he may use a range of search and select statements to find whether a ship exists which has, for example, its name on the sets INHARBOUR, SMALL and CRATED.

The second basic use of sets is to represent queues; this is possible as sets contain *ordered* lists of entity names. A selection process such as that described in the previous paragraph may well produce the result that several ships suitable for the berth exist in the harbour, and the user in this case may wish his program to select the first. In many real-life situations, the process of making a choice involves making a selection of a subgroup of objects which have suitable properties, combined with some priority and queuing considerations which guide the final choice of a single object. This state of affairs is reflected in CSL statements; for example, the

statement which selects the first ship waiting in harbour which is small and carries crated cargo and provides its name in the name type variable S, is

```
FIND S HARBOUR FIRST
     S IN SMALL
     S IN CRATED
```

The range of compound statements of this type is extensive, and is probably the most important feature of the language.

Time and dynamic descriptions

Interactions in a real system are dependent on time, and the system moves through time. It is therefore necessary to have some means of representing time in a simulation program. Time values are held in what are called T-cells. T-cells may arise in two ways; either defined as single integer cells or arrays or as cells attached to entities. In all cases they are recognized by prefacing their names with T. For example, if the class of ships is defined thus:

```
CLASS TIME SHIP. 100
```

then this serves to define entity names as above, and also 100 T-cells addressed as T. SHIP. 1, . . ., T. SHIP. 100 An array of integer T-cells could be defined as

```
TIME BREAKDOWNS (10)
```

T-cells have all the properties of other integer cells, and may participate normally in arithmetic and tests. Their time-advancing properties are additional.

The execution of a program is carried out under the control of a CSL executive routine in a repeated two-stage process as follows. Stage 1 treats all T-cells which arise in the program as if they contain time relative to "now" as zero; that is, a future time is positive. The T-cells are scanned to find the smallest positive non-zero value in any cell. This is regarded as the time of the next event or the time at which an event is next able to arise in the system. The program is now advanced to this position in time by subtracting this value from all T-cells. This completes stage 1.

In stage 2 the main routines of the program itself are entered. The user must specify his program as a series of individual routines called *activities*, and stage 2 consists of an attempt to obey each of the activities in turn. Each activity describes the rules relating to the performance of one kind of operation in the system: for example, that of unberthing a ship. The statements in an activity normally begin with a series of tests to find out whether the activity can be initiated: these may be tests on T-cells to see whether, for instance, any ships are due to leave a berth.

After the opening tests follow the statements which actually carry out the work of the activity; e.g. arithmetic and set-manipulation statements.

The actual question of division of the program into activities is governed by individual programming style. The activities must clearly cover all possible courses of action available in the system, but this is not the whole story. For instance, in the example in the Appendix the berthing of large and small ships are handled as separate activities. The orders in these are largely duplicate; should they therefore be combined into one activity? This is mainly a question of taste and as such is not easily resolvable on logical grounds.

The structure of a CSL test can now be more fully explained. The most frequent use of a test is at the start of an activity; under these circumstances, if the test fails, it may be assumed that the activity cannot be carried out and the natural thing to do is to try the next activity. For this reason the customary operation of computer test statements has been reversed in CSL; a test failure with no specified destination leads to transfer of control via the executive routine to the next activity, whereas in the case of success the next statement is obeyed. To provide more detailed control of flow a failure destination statement label may be specified, e.g.

```
DATA (10) EQ 4 @ 87
```

In the event of failure, control goes to the statement labelled 87.

It should be emphasized that stage 2 consists of an attempt to obey *all* the activities specified in the system. Apparently this involves much redundant effort, as at most points in time only one or two activities are likely to be entered successfully and the rest will be abandoned after a test or two, but a closer analysis shows that computing time to carry out work of this kind is expended in any simple simulation programming system, whether it is carried out under direct control of the programmer or not. It seems, therefore, more useful to make the necessary testing explicit and under the user's control.

When all the activities in the system have been entered, the normal control procedure is that a return to stage 1 takes place and time is further advanced. This procedure is not in itself sufficient; activities are interlinked, and the completion of one activity may render possible the initiation of another. For example, the unberthing of one ship will free a berth which may be used by another. The user can control this in two ways; firstly, by use of a special device (the RECYCLE statement) which causes further attempts to obey the activities to be made and secondly by careful choice of the order in which activities are specified.

Compilation

Simulation programs are notoriously hard to test on a computer. The reasons for this are quite fundamental; firstly a simulation program is usually a representation on a serial computer of processes which in real life take place in parallel and secondly, its course of operation is

dependent on random sampling techniques. The programming system must, therefore, offer to the user the maximum possible assistance in checking out his program. A system in which the user may have to resort to study of a machine dump or an assembly language listing of his compiled program is just not good enough in this particular area of programming.

The compiler which is at present available for the new CSL runs on an IBM 7094 computer under the IBSYS operating system. It carries out a thorough syntactic check on validity of an input program, and contains various facilities by which the user may request dynamic checking on, for example, array subscripts being within bounds during actual execution. Throughout the checking system the principle followed is that all reports to the programmer must be made strictly in source-language terms.

Sectors of program may be compiled and run independently. The problem of changing parameters in the program for re-runs, such as class populations or array dimensions, is solved by a programming device which enables such changes to be inserted without forcing re-compilation of the entire program.

Summary of facilities

The language provides facilities for describing complicated interactions between groups of objects which may have complex properties. The properties of objects can be expressed in two ways: by traditional numerical coding and by recording the names of objects sharing some common properties on sets, which are ordered lists of names. Facilities for the former approach resemble conventional languages, with some additional features such as statistical storage devices. The latter approach has two main features—firstly, it is possible to carry out sophisticated selection procedures using a series of language statements based initially on the predicate calculus, and secondly, queuing and priority systems can be constructed.

Time-values in the system are held in special integer variables regarded as holding relative times. The time-advancing mechanism depends on division of the program into activities. It is operated by an executive routine in two stages which alternate indefinitely; in stage 1 time is advanced, by treatment of the time cells, to the next event, and in stage 2 an attempt is made to obey every activity.

CSL is designed to express complex decision problems on a computer, by enabling the user to define the components of a system and to write orders describing complex interactions and choices amongst these components. It permits sophisticated handling of these interactions but it imposes the necessity for a fixed population of components; situations which involve flow necessitate a simple programming device, exemplified by use of the set OCEAN in the example as a source and sink for ships flowing into and out of the system. Most programming languages of this general type adopt a dynamic data structure which permits the user to imitate more directly the concept of flow, by, for example, writing orders which cause creation and destruction of entities. The disadvantage of a dynamic data structure is that program execution is usually less efficient and sophisticated, and for this reason CSL uses static data structures.

Conclusion

The uses of this type of programming language are manifold. Problems of on-line decision making and control are in their infancy, and the only noteworthy work yet done using CSL in this area is that by Laski at the Esso Petroleum Company. However, in the long run this may prove to be the biggest area of application of this type of language.

At present the language is mainly used as a tool to write simulation programs of new systems under design or old ones needing improvements or re-design. The many problems tackled have included traffic movement on two national railway systems and various other transport systems, industrial scheduling problems and a maternity hospital.

It will be realized that a special-purpose language such as CSL does more than provide its user with useful manipulation statements; it imposes a programming style and a way of thinking about the problem. So, for that matter, does a general-purpose language in a much smaller way; an ALGOL or FORTRAN user is constrained to think in terms of arrays and numbers, and a COBOL user in files and records. The basic problems of CSL style are these: deciding which classes of entities to define and which sets to define to record their properties in the most useful way and specifying an exhaustive list of activities to be programmed. These are mainly stylistic problems resembling those in arts other than the programming art; there are therefore no definitive solutions and every practitioner follows his own line convinced that all other approaches are inferior.

One final point needs emphasis. One can write, in CSL, a program which imitates a real system. By itself this step achieves nothing more than the production of a program; it must be preceded by the work needed to establish a correct description of the system under study, and it must be followed by use of the program as an experimental tool with which possible alterations or improvements to the design of the system can be studied. The actual preparation of a computer simulation is the central part of a complete study, but means nothing by itself.

Acknowledgements

I wish to make acknowledgement to IBM United Kingdom Ltd., who have supported this work on CSL, for permission to publish this paper.

I owe acknowledgement to many friends and colleagues, in particular to Messrs. Blunden, Grant and Parncutt of IBM United Kingdom Ltd., who played the major part in the production of the compiler, and to Dr. J. G. Laski, the co-author of the original CSL.

Appendix

Sample program: A port simulation

This example is a simulation of the operation of a simple port, which consists of an outer deep water harbour and a series of berths. Each berth can hold one large ship, which can only berth at full tide, or three small ships which can also move at half tide. The tide runs in a 12-hour sequence; out for 7 hours, half-tide for one hour, full tide for three hours and half-tide for an hour.

A distribution of unloading times for large ships is available as data, and unloading times for small ships are normally distributed. Inter-arrival times are negative exponentially distributed.

The program is to record the waiting times of large and small ships, and the times for which the berths are empty. The purpose of the simulation might be to study the operation as a basis for experiments to find a more efficient way of scheduling the working of the port, or to determine the effect that provision of extra berths would have. The scheduling used in this model is a simple first in—first out scheme.

```
PORT SIMULATION  EXAMPLE PROGRAM
CONTROL
CLASS TIME SHIP.100 BERTH.4
    DEFINE CLASSES OF 100 SHIPS AND 4
        BERTHS
SET OCEAN HARBOUR LARGE SMALL FREE
        PART FULL
SET SHIPIN(BERTH)
    DEFINE THE SETS REQUIRED, INCLUD-
    ING AN ARRAY OF AS MANY SETS
    AS THERE ARE BERTHS.  SHIPIN(X)
    WILL HOLD A LIST OF THE NAMES OF
    SHIPS IN BERTH X.
NAME S B
INTEGER TIDE TLARGE TSMALL
TIME CHANGE ARRIVE FINISH
    DEFINE TWO NAME VARIABLES, AN
    INTEGER VARIABLE TO SHOW THE
    STATE OF THE TIDE, AND ADDITIONAL
    TIME CELLS.  ALSO TWO INTEGERS TO
    HOLD TOTAL ARRIVALS OF LARGE AND
    SMALL SHIPS RESPECTIVELY.
HIST LARGEQ 25,2,5  SMALLQ 25,2,5
    IDLE 25,2,5
HIST UNLOD 20,3,5
    DEFINE THE HISTOGRAMS REQUIRED.
    LARGEQ HAS 25 CELLS WITH RANGES
    0-4 (MIDPOINT 2) ,5-9, 10-14 ETC.
    UNLOD WILL CONTAIN THE UNLOADING
    TIME DISTRIBUTION FOR LARGE SHIPS.
INITL
ACTIVITIES
TIDES ARRVL BTHL BTHS DBTH ENDNG
    SPECIFY THE LIST OF SECTORS
    (ACTIVITIES)
END

SECTOR INITL
T.FINISH=24000
T.CHANGE=7
T.ARRIVE=0
TIDE=0
    THIS SECTOR IS ENTERED ONLY ONCE
    AND SETS UP THE INITIAL STATE OF
    THE MODEL.  T.FINISH REFERS TO THE
    TIME AT WHICH SIMULATION IS TO
    FINISH, T.CHANGE TO THE TIME AT
    WHICH THE TIDE NEXT CHANGES AND
    TIDE SHOWS THE STATE OF THE TIDE
    AS FOLLOWS—0 TIDE OUT  1 HALF IN
    2 TIDE FULL  3 HALF IN  T.ARRIVE
    SHOWS THE TIME BEFORE THE NEXT
    ARRIVAL OF A SHIP AT THE PORT.
FOR X = 1,SHIP
    SHIP.X INTO OCEAN
FOR X = 1,BERTH
    BERTH.X INTO FREE
    T.BERTH.X=0
        INITIALLY ALL SHIPS ARE IN OCEAN
        AND ALL BERTHS FREE
READ (5, 10) UNLOD
    READ IN THE DISTRIBUTION GIVEN AS
        DATA.
10  FORMAT (14)
    END

SECTOR TIDES
    THIS SECTOR IS CONCERNED WITH
    TIDE CHANGES
T.CHANGE EQ 0
    WHICH CAN ONLY OCCUR WHEN THEY
    ARE DUE
TIDE+1
GOTO (10, 20, 10, 30) TIDE
    CHANGE TIDE MARKER AND RESET
    TIME CELL FOR NEXT CHANGE
10  T.CHANGE=1
    GOTO 60
20  T.CHANGE=3
    GOTO 60
30  T.CHANGE=7
    TIDE=0
60  DUMMY
    AND RETURN TO CONTROL SEGMENT
    END

SECTOR ARRVL
    THIS SECTOR IS CONCERNED WITH
    ARRIVALS OF SHIPS
14  T.ARRIVE EQ 0
    WHICH CAN ONLY OCCUR WHEN ONE
    IS DUE
```

```
    FIND S OCEAN FIRST &15
    S FROM OCEAN INTO HARBOUR
    T.S=0
        FIND THE FIRST SHIP IN THE OCEAN
        MOVE IT TO THE HARBOUR AND ZERO
        ITS TIME CELL
    T.ARRIVE=NEGEXP (7)
        SAMPLE THE TIME TO THE NEXT
        ARRIVAL
    UNIFORM (SYSTEMSTREAM) GT 0.75 &13
        S INTO LARGE
        TLARGE+1
    GOTO 14
13  S INTO SMALL
    TSMALL+1
    GOTO 14
        A QUARTER OF THE SHIPS ARE LARGE,
        OTHERS SMALL. GO BACK TO START
        OF SECTOR IN CASE NEGEXP HAS
        GIVEN A ZERO SAMPLE
15  WRITE(6, 100) T.FINISH,CLOCK
100 LINGEN
    1 NOT ENOUGH SHIPS IN MODEL —
      SIMULATION TERMINATED
    2 TIME LEFT ***** TIME ELAPSED *****
    T.FINISH = 0
        IF A SHIP IS NOT FOUND IN OCEAN,
        WRITE MESSAGE AND SET T.FINISH SO
        THAT SIMULATION CEASES IN SECTOR
        ENDNG.
    GOTO ENDNG
    END

    SECTOR BTHL
        THIS SECTOR IS CONCERNED WITH
        BERTHING LARGE SHIPS
    TIDE EQ 2
    FIND B FREE ANY
    FIND S HARBOUR FIRST
      S IN LARGE
        THE TIDE MUST BE FULL, THERE MUST
        BE A FREE BERTH AND A LARGE SHIP
        WAITING IN THE HARBOUR
    ENTER —T.S,LARGEQ
        WHEN THE SHIP ENTERED THE HAR-
        BOUR ITS TIME CELL WAS SET TO ZERO.
        SINCE THEN IT HAS BEEN REDUCED
        AT EACH TIME ADVANCE AND SO —T.S
        IS THE WAITING-TIME OF THE SHIP.
        THIS IS RECORDED IN THE HISTOGRAM
    B FROM FREE INTO FULL
    S FROM HARBOUR INTO SHIPIN(B)
        THE BERTH IS NOW FULL AND THE
        SHIP MOVES FROM THE HARBOUR
        INTO THE BERTH
    ENTER —T.B,IDLE
        JUST AS —T.S SHOWED THE SHIPS
        WAITING TIME SO —T.B SHOWS THE
        BERTH IDLE TIME
    T.S=SAMPLE(UNLOD)
        SAMPLE AN UNLOADING TIME FOR
        THE SHIP
    RECYCLE
        CAUSE ANOTHER PASS THROUGH THE
        SECTORS (BECAUSE MORE THAN ONE
        SHIP MIGHT BERTH AT THE SAME TIME)
    END

    SECTOR BTHS
        THIS SECTOR IS CONCERNED WITH
        BERTHING SMALL SHIPS. IT IS
        OMITTED FROM THIS EXAMPLE AS
        THE STATEMENTS IN IT ARE SIMILAR
        TO THOSE IN THE PREVIOUS SECTOR.

    SECTOR DBTH
        THIS SECTOR IS CONCERNED WITH
        DEBERTHING
    TIDE NE 0
        THE TIDE CANNOT BE OUT
    For X = 1, BERTH
        DEAL WITH EACH BERTH SEPARATELY
        IN TURN
20  FIND S SHIPIN(X) FIRST    &15
        T.S LE 0
        CHAIN
          S IN SMALL
          OR S IN LARGE
          TIDE EQ 2
    DUMMY
        FIND A SHIP IN THE BERTH WHICH IS
        READY TO LEAVE (TIME CELL HAS
        BEEN REDUCED TO ZERO OR BEYOND
        BY TIME ADVANCE) AND WHICH CAN
        DO SO AT THE PRESENT STATE OF THE
        TIDE. IF NONE — GO ON TO TRY THE
        NEXT BERTH
    RECYCLE
        SET RECYCLE SWITCH TO TRY SECTORS
        AGAIN BEFORE TIME ADVANCE (IN
        PARTICULAR BERTHING SECTORS MAY
        NOW SUCCEED)
      S FROM SHIPIN(X) INTO OCEAN
      S IN LARGE  &16
      S FROM LARGE
      T.BERTH.X= 0
      BERTH.X FROM FULL INTO FREE
      GOTO  15
        IF SHIP LEAVING IS LARGE BERTH IS
        NOW FREE. ZERO ITS TIME CELL SO
        THAT IDLE TIME CAN BE COMPUTED
        LATER. THEN GO TO NEXT BERTH.
16    S FROM SMALL
      SHIPIN(X) EQ 0  &17
      BERTH.X FROM PART INTO FREE
      T.BERTH.X=0
      GOTO  15
        SIMILARLY IF SHIP LEAVING IS SMALL
```

```
      AND NOW THERE ARE NONE LEFT IN
      THE BERTH.
17    SHIPIN(X) EQ 2  &20
      BERTH.X FROM FULL INTO PART
      GOTO 20
        IF SMALL SHIP IS LEAVING AND BERTH
        WAS PREVIOUSLY FULL, RECORD FACT
        THAT IT IS NOW ONLY PARTLY FULL
        IN EITHER CASE GO BACK TO SEE IF
        ANY MORE SHIPS ARE READY TO
        LEAVE THIS SAME BERTH
15      DUMMY
      DUMMY
      END
```

```
SECTOR ENDING
    THIS SECTOR IS CONCERNED WITH
    OUTPUT OF RESULTS
T.FINISH EQ 0
    WHICH IS TO BE DONE AFTER TIME
    HAS BEEN ADVANCED SO THAT T.
    FINISH HAS BECOME ZERO


THE REST OF THE SECTOR CONSISTS OF
OUTPUT STATEMENTS AND IS OMITTED.
STOP
END
```

References

BUXTON, J. N., and LASKI, J. G. (1962). "Control and Simulation Language," *The Computer Journal*, Vol. 5, p. 194.

CSL Reference Manual (1962), IBM United Kingdom Ltd. and Esso Petroleum Co. Ltd.

LASKI, J. G. (1965). "Using Simulation for on-line decision-making," presented at the NATO Conference on the role of digital simulation in O.R., Hamburg, September 1965.

CSL Reference Manual (1965), IBM United Kingdom Ltd.

Book Reviews

Switching Theory, by R. E. Miller. Vol. I, *Combinational circuits*, 1965; 351 pages. Vol. II, *Sequential circuits and machines*, 1966; 250 pages. (London and New York: *John Wiley and Sons Ltd.*, 98s. and 87s.)

There exists considerable uncertainty in the computer design world as to whether switching theory is properly a pursuit for mathematicians or engineers. Although there are those, including the reviewer, who believe that too mathematical a motivation is a widespread fault in the literature of this subject, it is an undeniable fact that progress cannot be made in it without recourse to a considerable range of mathematical methods. Probably only one previous book (Caldwell, 1958, *Switching circuits and logical design*, Wiley) can fairly be claimed to have given a thorough treatment from a largely engineering standpoint, and so much new material has emerged since then that there was a definite need for a more up-to-date text. Dr. Miller's two volumes appear to satisfy that need to a great extent, and whilst couched in quite rigorous mathematical form, nevertheless display strong awareness of the engineer's requirements.

In his preface the author states that the books were written primarily for "advanced undergraduate and graduate study" and if this is so we in this country can only marvel at and admire the high standard of knowledge in this rather specialized corner of the electrical engineering field expected of even advanced undergraduates in America. These volumes, particularly the second, contain details of ideas and methods which have appeared in technical journals only during the past two or three years, and which can by no means be considered as completely developed.

Volume I on combinational circuits commences with a general discussion of digital systems, as a sort of *hors d'oeuvre*, which is the best of its kind the reviewer has seen. Chapter 2 develops the idea of abstract Boolean algebras and shows how such an algebra having only two elements relates to the requirements of switching circuits. It includes also a brief discussion relating Boolean algebra to group and lattice theory, a section on graphical and cubical function representation, one on some special groups of functions (symmetric, unate, threshold) and one on functional decomposition. This chapter also is admirable both in coverage and treatment. The remaining three chapters deal thoroughly with normal form design, multi-output and multi-level circuits and bilateral networks, respectively. There are one or two grounds for criticism in this volume; first is the complete omission of any serious consideration of the important NOR and NAND universal decision functions, round which most modern computers are designed; second is the almost exclusive use in Chapters 3 and 4 of the somewhat unfamiliar cubical representation of functions. All the known methods of representation and minimization of functions are fundamentally similar and share a common failing in that the visualization of functions of more than a few variables is extremely difficult. If anything the cubical representation is worse than others in this respect and apart from a few special concepts (*e.g.* that of linear separability of threshold functions) does not appear to offer any great compensating advantages over, for example, a purely algebraic approach. Dr. Miller's preoccupation with the cubical approach causes him to give only cursory attention to the widely used graphical (*e.g.* Karnaugh map) and tabular (Quine-McCluskey) procedures.

This volume contains a number of typographical errors, which are not serious, and a few mistakes and omissions sufficiently serious to cause temporary confusion. For example on page 33 the symbol x_o is suddenly redefined as a dividend after it has been used for three pages to mean the sign bit of a binary number. Again on pages 110 and 111, two partially ordered lattices are apparently upside down, since the vertices quoted as being least are in both cases at the top. A final example is on page 146, where ø is used without

(*Continued on p.* 166)